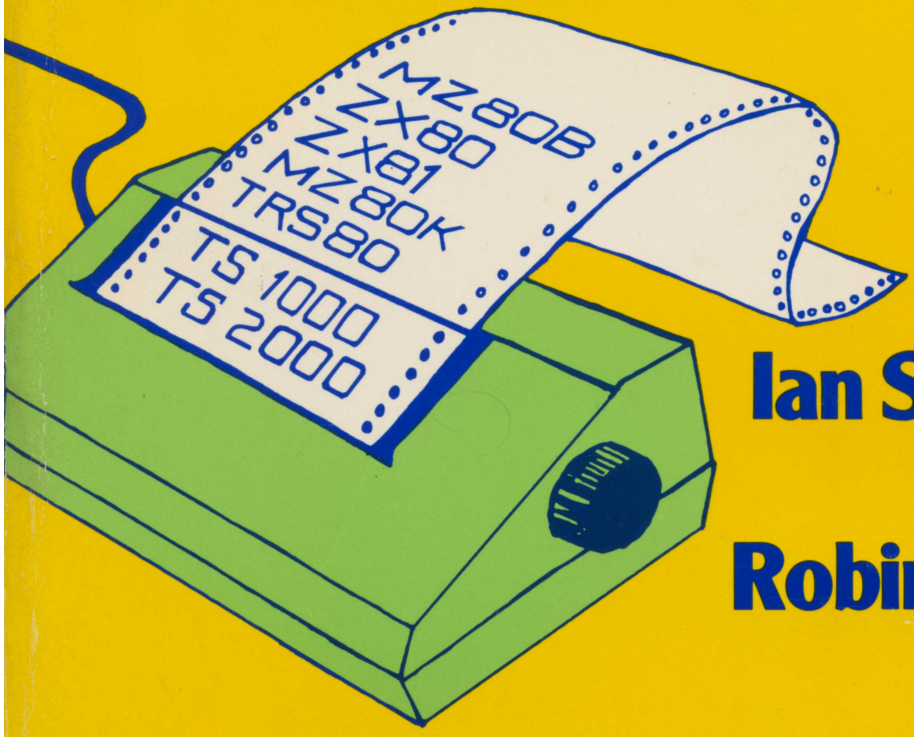
Machine Code and better BASIC



Ian Stewart
and
Robin Jones

Machine Code and better BASIC

Ian Stewart
Robin Jones

Birkhäuser
Boston • Basel • Stuttgart

This is the American edition of the British publication, "Machine Code and better Basic," which was published by Shiva Publishing, Ltd. This American edition has been adapted to the American market, taking into account new developments in technology and current references.

Library of Congress Cataloging in Publication Data

Stewart, Ian.
Machine code and better BASIC.

Bibliography: p. 184
Includes index.
1. Sinclair ZX81 (Computer)—Programming.
2. BASIC (Computer program language) I. Jones,
Robin. II. Title.
QA76.8.S625S83 1983 001.64'24 82-24319
ISBN 3-7643-3115-1 (Switzerland)

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission of the copyright owner.

© Ian Stewart and Robin Jones, 1983
ISBN 3-7643-3115-1
Printed in USA

The Rubáiyát of Programmer Khayyám

Awake! For Morning's fickle hand doth load
Updated software in the daylight mode.
    Return from sluggish subroutine of night:
DIM the array, but brilliant the code!

Myself when young did frequently frequent
The data-punching rooms, and heard great argument;
    But evermore it seemed I must emerge
By that same interface wherein I'd went.

Ah, but my computations, people say,
Process the text to clearer meaning? Nay,
    Though Man may seek the symbols to construe
The Greater Editor will have his way.

The User programs while the disk-drives whisk;
Taps the mad keyboard of a mind at risk.
    The work of years comes suddenly to naught
As random noise corrupts the floppy disk.

Some for the glories of this world, and some
Sigh for a pointer to the world to come.
    Ah, seize the output, let the record go,
Nor heed the rumble of magnetic drum!

A User-Manual 'neath a labelled tree,
A pint of beer, a ploughman's lunch—and Thee!
    What care I then for megabytes?
Thy tiniest bits yield megabytes for me.

The moving cursor writes, and having writ
Moves on: nor all your piety nor wit
    Shall lure it back to cancel half a line
Nor all your Tears wash out a word of it.

But wait! say ye: The console's cursor keys
Can Backspace, Rubout, Edit as we please?
    Not so! These merely tidy the display:
Still the grim input's in the memories.

Some peek the ROM of Time's predestined flight;
Some seek within Life's RAM new lines to write.
    In vain each strives t'assemble faultless code,
For still Death's Digits poke the final byte.

Contents

	Preface	ix
Data Structures		1
1	Arrays	3
2	Searches	12
3	Stacks	15
4	Queues	20
5	Linked Lists	25
6	Trees	31
7	Game Trees	36
Structured Programming		47
8	Zedtext	49
9	Checkout	69
10	French Countdown	82
Machine Code		103
11	Numbers in Machine Code	107
12	Positive and Negative	111
13	Machine Architecture	114
14	Jumps and Subroutines	118
15	Indirection and Indexing	120
16	At last the Z80!	124
17	Load	126
18	Arithmetic	128
19	A Subset of Z80 instructions	133
20	A Machine Code Multiplier	141
21	The Display File	145
22	Some Things I haven't told you	152

Appendices 165
1 Hex/Decimal Conversion 166
2 Memory Reservation Tables 167
3 System Variable Addresses 168
4 Summary of Z80 Commands 169
5 Z80 Opcodes 171
6 HELPA 174
7 DOWNLOAD 182

Bibliography 184
Index 185

Preface

In a way, this is a sequel to our book ***Timex/Sinclair 1000 Programs, Games and Graphics,*** which henceforth we shall refer to simply as ***Timex/Sinclair 1000.*** It's designed to do two things: exploit the extra power of a 16K-expanded memory, and introduce the delights of Z80 Machine Code.

The principles of good programming are machine-independent, and almost everything in this book applies (subject to minor changes to system variables) to any machine built around a Z80 or Z80A CPU: Timex/Sinclair 2000, the Sharp MZ80B or MZ80K, the Tandy TRS80, and the Research Machines 380Z, for example. Owners of these machines will easily adapt the descriptions given; but for definiteness we have chosen to write for the Sinclair ZX81. There is one overwhelming reason for this choice.

As we write, there are over 400,000 Sinclair ZX81s (essentially identical to the ***Timex/Sinclair 1000***) in existence. This "motorbike of computers" has brought computing into the home in a big way: people who would never have dreamed they would even be *using* a computer now *own* one—and use it to educate their children, balance the household accounts, work out their taxes, play games, and run electric trains or photocopiers.

This book is for those who have mastered the basics, and want to cut their teeth on something more substantial.

One thing you can do is write more complicated BASIC programs—after all, that's why high-level languages like BASIC were invented. To write bug-free programs which can be understood and modified without courting the lunatic asylum, you need to develop a well-structured style. We pursue this line of thought in the sections on *Data Structures* and *Structured Programming*.

You can also go outside BASIC altogether. On the ***1000*** that means *Machine Code*: the language that the Z80 microchip talks. Often (though not always) Machine Code is faster than BASIC: the snag is that you have to do more of the thinking yourself by way of compensation. We didn't think that you'd want to write *long* Machine Code routines at this stage; but we've given enough information for you to take several steps along that road, so that the more advanced books (listed in the bibliography) become accessible.

One important feature of Machine Code is that it teaches you a lot more about the way the computer actually *works*—and in the coming age of microeverythings, that's going to be an important piece of knowledge.

We've included plenty of programs as examples: searching a stock list, enqueuing and dequeuing data, pushing on to a stack and popping from one, cataloguing a library, a game-tree where the computer works out its own winning strategy, a complete text-editor (or word-processor), a simulation study of supermarket checkouts which applies equally well to allocating hospital beds, and an educational program for testing French vocabulary.

There are Machine Code routines to give the display a checkerboard pattern (instantly), or turn it into inverse video (in a flash); to draw lines of characters, to add

and multiply numbers, to move data around in RAM, to scroll limited regions of the screen fast enough to be useful, to scroll sideways, and to renumber BASIC lines.

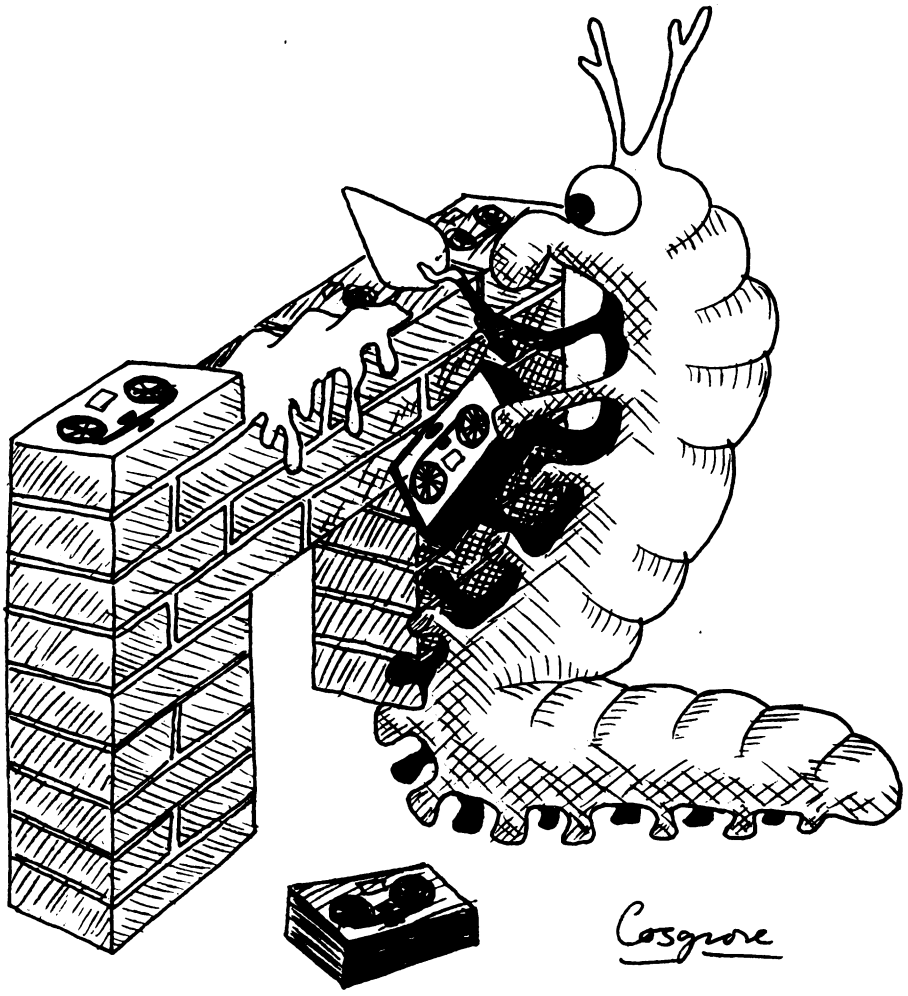
The appendices include several tables of useful data: hex/decimal conversion (including 2's complement notation for relative jumps); memory reservation; system variable addresses; a summary of Z80 commands; a list of all 694 of the Z80 Opcodes. There's a modestly powerful BASIC program called HELPA (Hex Editor, Loader, and Partial Assembler) that takes most of the pain out of Machine Code; and an accessory called DOWNLOAD which will let you incorporate a Machine Code routine into a program loaded from tape (thus *passing routines from one tape to another* via the ***1000***).

If you want, you can copy out most of this material and RUN it without worrying too much about why it works. But what we *hope* you will do is work through the explanations too. That way, you'll be learning how to write your own programs and how to get out of the 16K RAM some of the marvellous things it is capable of.

After all, you want the ***Timex/Sinclair 1000*** to know who's boss.

So far, we've referred to ourselves as "we"—but as in *Timex/Sinclair 1000*, we found this didn't always work out later. So from now on, we'll refer to ourselves in the singular, as "I". Whenever we say "we", we'll mean "I and the reader". It may sound a silly idea, but it's actually more informative that way.

Data Structures



There's a lot more to programming than just knowing the features of a particular programming language. For one thing, we need a clear idea of the procedure, or *algorithm*, we are going to employ in solving a given problem. We need to develop ways of splitting this algorithm up into chunks small enough to be convenient for coding in BASIC, or whatever language we're using. That's a topic we'll come back to. For the minute, I'm concerned about an even more fundamental problem in program design: that of the organization, or structure, of the data which the program is to manipulate. Actually, we're quite used to the idea of imposing a structure on data which we wish to handle manually, although we probably wouldn't give the operation such a grand title. Think about a bank statement for example:

Date	Details	Debits	Credits	Balance
4/Ø7	B/FWD	—	—	148.78
5/Ø7	897669	24.48	—	124.3Ø
9/Ø7	S. TILL	2Ø.ØØ	—	1Ø4.3Ø
1/Ø8	W.L.LTD	—	254.11	358.41

It's clear that the data presented here are structured in a very formal way. Each type of data—date, debit, credit and so on—only appears in a given column. There's even some implied organization—we know, without being told, that the first number in the date is the day, and the second is the month. (This implicit structure can get us into trouble—if the date is read by an American he'll see it the other way round!) Of course, somebody had to decide in the first place that this was a convenient way to present bank statements and, naturally, he would have asked the question: "How easy is it going to be to deduce useful information from this structure?" We, on the other hand, are not interested in deriving information directly from a data structure, but in getting the computer to derive the information for us. So we are always going to have to ask the question: "How easy is it going to be to write programs to derive useful information from a particular data structure?"

So it's clear, I think, that it's important to have the data organized before you start to think about the program.

We're going to look at a number of common data structures in this book, and to describe some of the ways in which they are used. The list isn't exhaustive, and in some cases it can be useful to design a structure which is quite novel; so don't be put off from doing so simply because a structure which seems convenient isn't listed here!

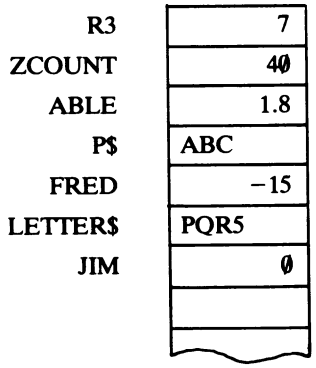
Sometimes in this section I'm going to use "illegal" names for variables—for example there's an array called BIGMAP and a string called MNAME$. Some BASICs allow this, but the version used by the ***1000*** doesn't. When you key in the programs, replace these illegal names by their first letters (unless already used in the program, in which case it's up to you). The aim is to improve the clarity of the *explanation* of the programs by giving a built-in reminder of the variable's meaning. I've reminded you the first few times this convention is used.

Arrays are directly available in BASIC and easy to use: for example, to list commercial data. But there are other ways to use them—including the analysis of weather-maps.

1 Arrays

Some structures are built into languages, and some you have to build for yourself. In BASIC, there is only one data structure within the language. It's called an *array*, and it seems a convenient place to start.

Let's review the way we think about the organization of the computer's memory. We could see it as a series:



of cells, each able to contain a number or string of characters, and each given a name, as in the diagram, for example.

This is fine for many applications, but not for all. For instance, suppose we want to input 200 numbers and hold all their values at the same time. It's no good writing:

```
10  FOR P = 1 TO 200
20  INPUT X
30  NEXT P
```

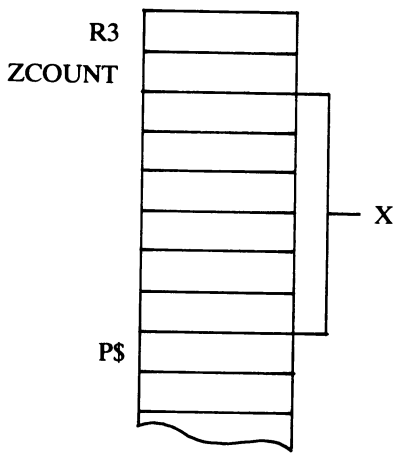
because with every loop, a new value will be entered into X, wiping out the old one. So, in the end, only the final value in the sequence will be in memory.

We could write:

```
10  INPUT X1
20  INPUT X2
30  INPUT X3
40  INPUT X4
    .............
2000  INPUT X200
```

but it would be a pain, and anyway this will occupy a lot of memory within the program

An array provides us with a way out of the dilemma. BASIC allows us to specify a whole block of memory as having just one name:



The block of memory represented above is called the *array* X. (It can have any valid BASIC name.)

An array may have any number of memory cells in it, but we have to tell the interpreter, at the beginning of the program, how many cells to allocate to it. This is done with a DIM (short for dimension) statement. In the above case, X has 6 cells, so we would write:

```
10 DIM X(6)
```

We still need to be able to refer to individual cells, or *elements*, within X. BASIC allows us to talk about X(1), the first element of X; X(2), the second; X(3), the third; and so on.

Coming back to our original problem, we could write:

```
10 DIM X(200)
20 INPUT X(1)
30 INPUT X(2)
40 INPUT X(3)
etc., etc.
```

"Hang on!" You're all saying, "That's no better than it was before. In fact, it's worse, because there are loads of extra brackets."

True, true. However, the trick is that when we wrote X1, X2, X3 etc., the 1, 2 and 3 are part of the variable names and can't be altered; but if we write, for instance, X(P) then the computer sees this as X(1) if P = 1, X(2) if P = 2 and so on. So the value in the brackets (called a *subscript*) can be changed.

In this case, we want to change the subscript by starting it at 1, and adding 1 to it until it reaches 200, and that's a clear cue for a FOR loop.

```
10 DIM X(200)
20 FOR P = 1 TO 200
30 INPUT X(P)
40 NEXT P
```

Why should we want to store 200 numbers all at once? About the simplest program I can think of in which it's clearly necessary, is one to print in reverse order the numbers input to it:

```
10 DIM X(200)
20 FOR P = 1 TO 200
30 INPUT X(P)
40 NEXT P
50 FOR P = 200 TO 1 STEP -1
60 PRINT X(P)
70 NEXT P
```

Obviously we can't print anything until the last number has been read, and we must remember all the previous numbers, since they are to be printed subsequently. A more practical example is the Bubble Sort. (***Timex/Sinclair 1000***, p. 78.)

CALCULATING DISCOUNTS

Here's another example of a rather different nature.

Suppose we run a wholesaling business and we split our customers into the following groups, each of which is offered a different discount, shown in brackets:

1. Private (0%)
2. Local authority (4.0%)
3. Education (6.5%)
4. Government (7.0%)
5. Trade (9.0%)
6. Trade—special contract (12.0%)

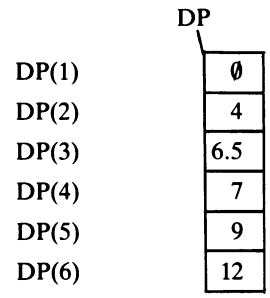
What we would like is a program that will accept a customer's name, his type (1–6), the number of items he's buying and the retail value per item, and print out the details of his bill.

Since the calculations depend on the customer type we might expect to have a series of IF statements like:

```
50 IF TYPE = 1 THEN . . .
60 IF TYPE = 2 THEN . . .
70 IF TYPE = 3 THEN . . .
```

This would be tedious but not beyond reasonable bounds. But what if there were 300 categories? . . .

Let's set up an array called DP (for Discount Percentage) like this:



Actually, on the ***1000*** you can't use more than one letter for an array; but as I said above, for clarity I'm going to anyway. If you want to write a program, change DP to D, or some other suitable letter.

Now, if TYPE = 1, the discount value we're interested in is in DP(1). If TYPE = 2, the value we want is in DP(2). In other words the value we are after is always in DP(TYPE)!

So the code to input the data and evaluate the discount would look something like:

```
80  INPUT N$
82  INPUT TYPE
84  INPUT NUMBER
86  INPUT VALUE
90  LET TOTAL = NUMBER * VALUE
100 LET DISCOUNT = TOTAL * DP(TYPE)/100
```

No IFs anywhere!

Note that we're using TYPE to point to the right value in the array. We call a variable used in this way a *pointer*.

How can we set up the array in the first place? The simplest way is to have a series of assignment statements like:

```
5  DIM DP(6)
10 LET DP(1) = 0
15 LET DP(2) = 4
20 LET DP(3) = 6.5
25 LET DP(4) = 7
30 LET DP(5) = 9
35 LET DP(6) = 12
```

This is OK for small arrays but tedious for large ones. An alternative is to use an input loop:

```
5  DIM DP(6)
10 FOR P = 1 TO 6
20 INPUT DP(P)
30 NEXT P
```

but, of course, we don't want to execute this routine every time the program is run, although we may wish to do so occasionally, if the discounts change. So we could have:

```
1  PRINT "NEW DISCOUNTS? (Y/N)"
2  INPUT A$
3  IF A$ = "N" THEN GOTO 80
```

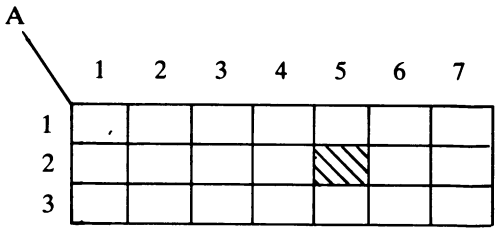
so that the program jumps round the input routine if the discounts aren't changed. When you save the program, the array values are also saved, so this will work for successive runs provided you key in GOTO 1 and not RUN (RUN clears all variable values).

ARRAYS IN TWO DIMENSIONS

What I've been describing so far is called a *one-dimensional array* or *vector*. It's possible to have a *two-dimensional array* or *table*. We specify such an array by a DIM statement as before. For example:

```
10 DIM A(3,7)
```

specifies an array which looks like:



In other words, I've identified for the computer a table having 7 columns and 3 rows. There isn't, though, any standard convention which decrees that the number of columns must appear before the number of rows in the DIM statement. It's the way you think about the table that matters. In other words, I could, equally validly, have specified the same array by:

```
10  DIM A(7,3)
```

provided that, in each subsequent reference to the array, I always give the row number second, and the column number first.

The golden rule is: having chosen a convention, stick to it; you're less likely to make a mistake. In this book I'll use what is probably the commonest convention: give a row number first, the column number second. That's convenient because one-dimensional arrays are usually drawn as single columns.

Now, when we want to refer to a particular cell in the table, A, we can do so by specifying the row and column which intersect at that cell; so if we wanted to set the shaded cell in the diagram to 24, we could write:

```
50  LET A(2,5) = 24
```

OK; that's the basic idea.

Now, let's put it to some use.

Imagine that you are a geographer, who wants to keep a record of the annual rainfall in a particular region. There are weather stations dotted around the region, so we might have a map which looks like Figure 1.1, in which the region of interest is shown shaded, and a star denotes a weather station.

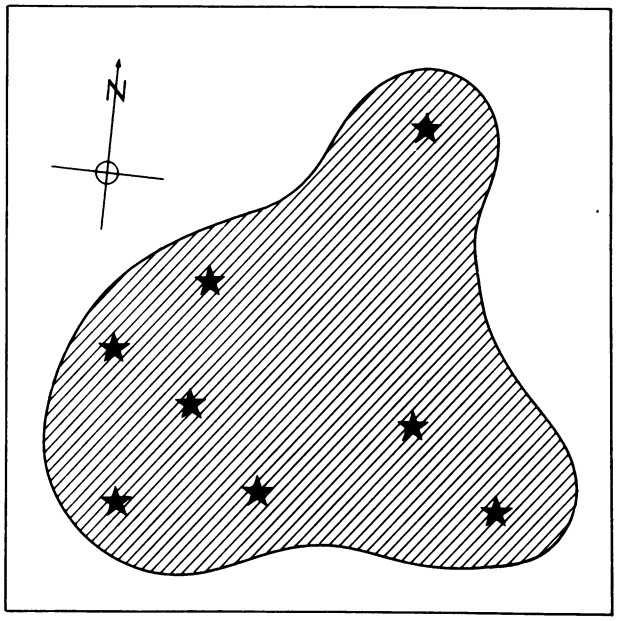


Figure 1.1

Obviously, the map is two dimensional, and we can easily represent it in a corresponding two-dimensional array by giving each point on the map a numeric code. There are only three types of point:

1. Points inside the region not at a weather station.
2. Weather stations.
3. Points outside the region.

At the weather stations we know the rainfall so the corresponding position in the array can just contain this value (in millimetres). Two values which can't be confused with a rainfall value are now needed to identify points inside and outside the region. Since you can't have negative rainfall, we could choose −2 for a point outside the region and −1 for one inside. So an array which models the map would look something like:

```
−2   −2   −2   −2   −2   −2   −2   −2
−2   −2   −2   −2   −2   461  −2   −2
−2   −2   −2   −2   −1   −1   −2   −2
−2   −2   401  −1   −1   −1   −2   −2
−2   400  −1   −1   −1   −1   −2   −2
−2   −1   440  −1   −1   480  −2   −2
−2   420  −1   424  −1   −1   484  −2
−2   −2   −2   −2   −2   −2   −2   −2
```

Of course, the resulting shape doesn't look as smooth as the original; in fact, it looks rather like a CEEFAX weather map, which is no big surprise, since they are created in much the same way. Anyway, we can always improve the appearance (and accuracy) of the representation by making the array bigger.

OK, we've managed to store a representation of a map in the computer. So what? What's wrong with a conventional atlas? Nothing, except that a conventional atlas leaves you to deduce whatever information you need from it. Now that our representation is inside the computer's memory we can write programs to answer all sorts of questions about the map:

How many weather stations are there?
What percentage of them have rainfall figures over 410 mm?
What is the highest recorded rainfall, and where is it?
What is the area of the region?

With a map of practical size, it would be very tedious to answer any of these questions manually; let's see how simple it would be to write the programs.

We'll suppose that the array has been set up by:

```
10  DIM MAP(50, 50)
```

(again on the ***1000*** you'd have to replace MAP by, say, M to get the syntax right) and that the values have already been input to it. We want to know how many stations there are.

So a rough outline for the procedure would be:

- Examine a cell.
- If it contains a non-negative value it's a weather station so count it.
- Repeat the process for all cells.

For a particular cell, somewhere in MAP, whose column value is C and whose row value is R, the IF statement we need is:

```
1030  IF MAP (R, C)>= 0 THEN LET NWS = NWS + 1      [NWS = number of
                                                     weather stations]
```

To deal with all the columns from 1 to 50 in a row R, we need a FOR loop:

```
1020 FOR C = 1 TO 50
1030 IF MAP (R, C)>= 0 THEN LET NWS = NWS + 1
1040 NEXT C
```

and to deal with all the rows from 1 to 50, we need a FOR loop round that:

```
1010 FOR R = 1 TO 50
1020 FOR C = 1 TO 50
1030 IF MAP (R, C)>= 0 THEN LET NWS = NWS + 1
1040 NEXT C
1050 NEXT R
```

If you've read ***Timex/Sinclair 1000*** you'll probably remember this kind of "nested" FOR loop construction (I used it to build a graphics rectangular block).

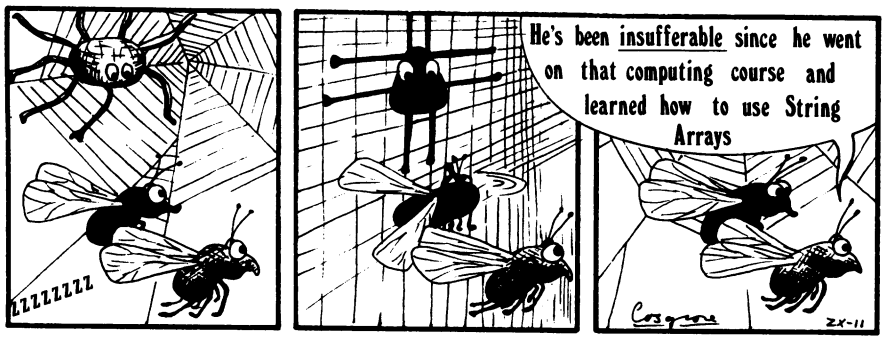
Now all we need to do is make sure that NWS contains zero to begin with:

```
1000 LET NWS = 0
```

and print the result out at the end:

```
1060 PRINT "NO. OF STATIONS = "; NWS
```

The other three problems I listed are very similar. They all require the same pair of FOR loops. The IF statements, and what to do when they are true, change, of course. You might like to try them.



ARRAYS IN THREE DIMENSIONS

Let's suppose that our geographer wants to use a bit more raw data. He's now got rainfall figures for each month at each weather station.

So we need twelve two-dimensional arrays like MAP to represent these data. Each array is like a page of an atlas. Shouldn't we be able to combine the arrays, like binding the atlas pages into a book? Well, we can. What we end up with is a three-dimensional array, which we could define by:

```
10 DIM BIGMAP (50, 50, 12)
```

(replace BIGMAP by B in an actual program, as before) and the third value in the brackets represents the number of months.

We can ask all the same questions as before, and the only change in the programs will be that we need another, outside, loop to change the month value.

For instance, the new loop could be:

```
FOR MONTH = 1 TO 12
.............
NEXT MONTH
```

and the IF statement will look like:

```
IF BIGMAP (R, C, MONTH) = . . . THEN . . .
```

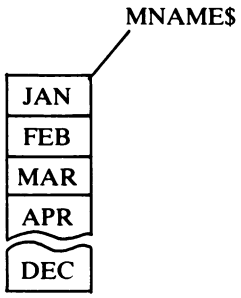
(Of course, some questions, like "How many stations are there?" don't change with time, and if you try this with that program you'll just get 12 times as many stations as you should.)

How about a routine to tell us the rainfall figure for a particular station in a given month? The code could look like this:

```
2000  PRINT "ENTER MAP REFERENCE OF STATION (R, C)"
2010  INPUT R
2020  INPUT C
2030  PRINT "ENTER 1ST 3 LETTERS OF MONTH"
2040  INPUT M$
2050  IF M$ = "JAN" THEN LET MONTH = 1
2060  IF M$ = "FEB" THEN LET MONTH = 2
2070  IF M$ = "MAR" THEN LET MONTH = 3
.............
and so on
.............
2160  IF M$ = "DEC" THEN LET MONTH = 12
2170  PRINT "REQUIRED FIGURE IS:"; BIGMAP (R, C, MONTH); "MM"
```

Lots of IFs again! How can the chunk of code between lines 2050 and 2160 be improved?

Let's set up an array of month names (much as we set up the discount table earlier). We'll call it MNAME$ although on the ***1000*** you'd need to find a single-letter name, such as N$.



and then replace the relevant lines with:

```
2050  FOR MONTH = 1 TO 12
2060  IF MNAME$ (MONTH) = M$ THEN GOTO 2170
2070  NEXT MONTH
```

Get it? It searches through the MNAME$ array looking for a match with the input

month (M$). When it finds one, the value of the pointer, MONTH, is the numeric equivalent. A good maxim is:

> If you've just written a piece of code with a lot of similar IF statements, one after the other, there was probably an easier way.

ARRAYS IN HIGHER DIMENSIONS

Perhaps you noticed that I carefully refrained from saying "A three-dimensional array looks like a cube" in the preceding section. Of course, you can think about it like that, but it makes it difficult to think about what a four-dimensional array (yes, you can have one!) behaves like, because it doesn't *look* like anything. We can't imagine four-dimensional solids. But that doesn't imply that it can't mean something.

At the risk of flogging a dead geographer, let's return to our map problem, and suppose that now he provides us with the monthly figures for a decade! So we now have ten 3D arrays. Why not call that one 4D array, just as we called the twelve 2D arrays one 3D array. Now we refer to a particular cell in the array as:

EVENBIGGERMAP (R, C, MONTH, YEAR)

and we only have to provide the appropriate values of R, C, MONTH and YEAR to elicit the required data.

In principle, there's nothing to stop us having five-, six-, seven- or even higher-dimensional arrays. Each extra dimension is used to store another attribute such as population, death rate, temperature and so on.

In practice, we're limited by the available memory. Actually, as far as a 16K ***1000*** is concerned, we had already run out of memory at the 3D stage. It's easy to calculate:

We had a 50 × 50 × 12 = 30,000 cell array.

Each cell occupies 5 bytes, so we're using 150,000 bytes. A 16K memory has 16 × 1024 = 16,384 bytes. Oops!

Of course, we can always make the map smaller to fit it in; 16 × 16 × 12 just fits (but by the time the program is stored as well, it might not!). So these refinements are really for larger machines, although there can be occasions when a 7 × 7 × 7 × 7 4D array (which is getting on for the largest possible in 16K) is useful.

Given a list of data, how do you find the item you want? You *could* just read through the whole list . . . but there may be better ways.

2 Searches

It's obvious that one of the most useful things we can do with an array is to search it for some particular piece of information; in fact, we've already done so, in our examples, several times. On each previous occasion, we've shown what is known as a *linear search*, so called because every element of the array is examined in turn until the target cell is reached. This is fine for small arrays, but with large ones it can be rather time-consuming.

There's an alternative, known as a *binary search*, which is faster, provided the data are in a known order in the array. Let's look at an example. Remember my wholesaler with his discount system? He's now getting ambitious. He wants to implement a stores control system. Each item he deals with is given a reference number, and the information held about each item is the number in stock and the price per unit.

So we could set up an array like:

STOCK

Ref. No.	No. in stock	Price
1384	58	31.72
1791	246	2.60
2114	15	254.00
2164	2486	0.53
8561	1418	0.16

Note that the reference numbers are in ascending order. The procedure won't work otherwise.

Among other routines, we would like one which allows the storeman to enter an item reference number, and the system responds by displaying the number currently in stock, and the price of the item.

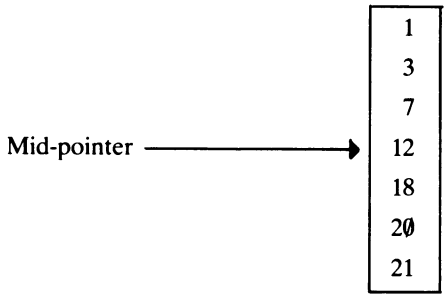
The way a binary search does this is first to look at the middle row in the table. Thus if there are 100 rows, we look at the 50th. (As there isn't an exact middle row if the number of rows is even, the 51st will also work, if you prefer.) If the reference number we're looking for is greater than the one we find here, it can't be in the lower half of the table, so we discard that, and concentrate on the upper half. Now we repeat the process for the remaining part of the table. This will chop out one half of the remainder, and continued repetition will ultimately narrow the search to one row.

Sounds long-winded? Well, think about an example. Suppose there are 1000 items in the table. According to Murphy's law the one you're after is always the last one you look at, so a linear search requires an examination of all 1000 items. A binary search removes:

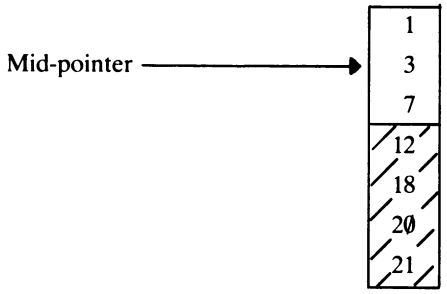
500	items	from	consideration	on	the	1st	test
250	"	"	"	"	"	2nd	"
125	"	"	"	"	"	3rd	"
62	"	"	"	"	"	4th	"
31	"	"	"	"	"	5th	"
15	"	"	"	"	"	6th	"
7	"	"	"	"	"	7th	"
3	"	"	"	"	"	8th	"
1	"	"	"	"	"	9th	"

which means we must find the target on the 10th test (even with Murphy's pessimism)!
The diagram below may make the idea a little clearer.

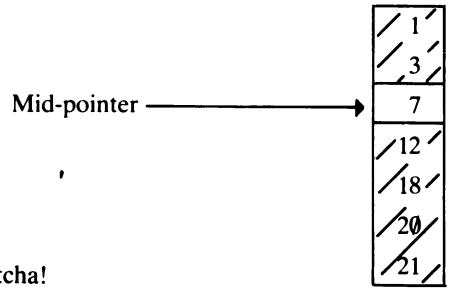
Target:7



12 > 7, so ignore the top half of the table.



3 < 7, so ignore the bottom half of the table



Gotcha!

Let's write a routine to do this for the STOCK array. (Again we use an illegal name for clarity: replace it by S in a program.) We need three pointers; one to the top of the region we're examining (TP), one to the bottom (BP), and the mid-pointer (MP), which is just (BP + TP)/2. So for an array with 1000 rows we start with:

```
2500  LET BP = 1
2510  LET TP = 1000
2520  LET MP = INT ( (BP + TP)/2)
```

The INT is strictly necessary because otherwise MP may include a decimal part (as here: 1001/2 = 500.5) which doesn't make sense.

Now we need to compare the required reference number (assuming this has already been entered into RN) with the reference number in STOCK pointed at by MP:

```
2530  IF STOCK (MP, 1) = RN THEN GOTO 3000
```

(Remember that item reference numbers are all in column 1.)

So when we get to line 3000, MP is pointing to the row we want and we'll be able to write:

```
3000  PRINT "NO. IN STOCK IS"; STOCK (MP, 2)
3010  PRINT "PRICES"; STOCK (MP, 3)
```

If the condition isn't met, however, we need to know if the value pointed to by the mid-pointer is greater than the target value and, if so, we chop out the top half of the table:

```
2540  IF STOCK (MP, 1) > RN THEN LET TP = MP - 1
```

On the other hand, the value pointed to by MP might be less than the target value:

```
2550  IF STOCK (MP, 1) < RN THEN LET BP = MP + 1
```

Now we need to calculate the new mid-pointer so:

```
2560  GOTO 2520
```

Tying the whole thing together gives:

```
2500  LET BP = 1
2510  LET TP = 1000
2520  LET MP = INT ( (BP + TP)/2)
2530  IF STOCK (MP, 1) = RN THEN GOTO 3000
2540  IF STOCK (MP, 1) > RN THEN LET TP = MP - 1
2550  IF STOCK (MP, 1) < RN THEN LET BP = MP + 1
2560  GOTO 2520
3000  PRINT "NO. IN STOCK IS"; STOCK (MP, 2)
3010  PRINT "PRICE IS"; STOCK (MP, 3)
```

Of course, there are lots of refinements you could build in. What happens if you enter a nonexistent item reference number, for instance? I'll leave you to think about that one.

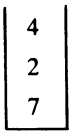
"Last in, first out" is the rule for a stack, either in the Tower of Hanoi or in a machine code program. Here's how to build an efficient one.

3 Stacks

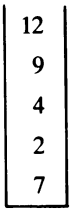
As I've said before, arrays are the only data structures inside BASIC. Other structures have to be built, usually in terms of arrays. So let's look first at a structure which is very simple to implement in terms of an array, called a *stack*.

The name "stack" explains its operation pretty well. You can only pile things on top of it, and you can only remove items from the top.

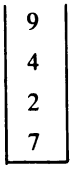
For example, we might start with:



If we add 9 and 12 (in that order) to the stack we get:

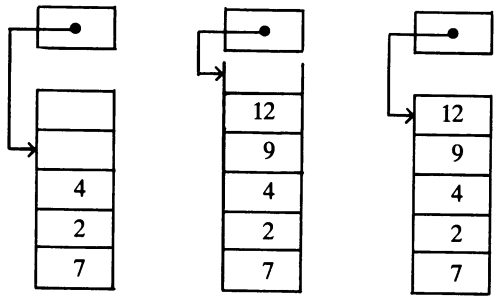


Remove an item from the stack and we have:



Obviously what we have is very like a one-dimensional array, but with the added restriction that access may only be made at a particular place (the top of the stack). Unfortunately, the top moves about inside the array depending on how many items are on the stack. So we need a pointer to determine where the top of the stack is. We'll define it to point at the first free location on the stack.

So now the previous example looks like:



Notice that when the 12 is removed from the stack it does not have to disappear from the array, because the pointer tells us where the top is to be considered to be.

STACK ROUTINES

OK. Let's try writing BASIC routines to implement a stack. First of all, what routines are needed? Well, there are three:

1. Set up the stack in the first place. We'll call this INITIALIZE.
2. Load a value on to the stack. The technical term for this is PUSH.
3. Remove a value from the stack. This is called POP.

The INITIALIZE routine can be put right at the beginning of the program:

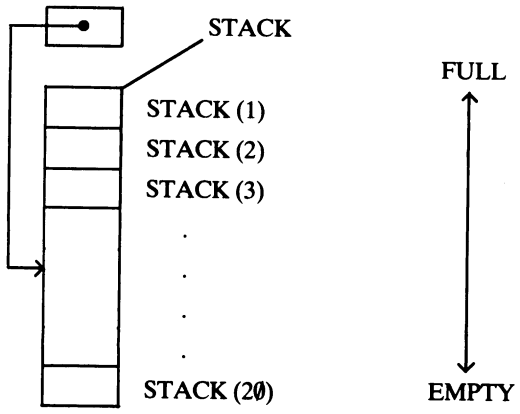
```
10 DIM STACK (20)
20 LET SP = 20
```

For a stack to hold a maximum of 20 items that's all it is! Putting the stack pointer (SP) to 20 tells us the stack is empty because it defines STACK (20) to be the first free location.

The other two routines will be needed throughout the main program and so they will be subroutines. PUSH looks like:

```
5010 LET STACK (SP) = V
5020 LET SP = SP - 1
5030 RETURN
```

(assuming that V contains the value to be PUSHed). What's happening is that the value is put into the array where SP is pointing. As that is the first free location, that's fine. Then the stack pointer is moved to point to the next free location. One is subtracted from it because of the mental model I have of the stack:



(There's nothing to stop you thinking about it the other way round if you prefer; provided, of course, you do it consistently.)

Actually, this diagram gives us a clue to something we haven't considered: what happens if the array fills up? When this happens, SP will contain zero, and an attempt to execute line 5010 will lead to an error message (report code 3). So we need a line 5000 to test for this condition:

```
5000  IF SP = 0 THEN GOTO 5040
```

and:

```
5040  PRINT "STACK FULL"
5050  STOP
```

POP will be similar:

```
6000  IF SP = 20 THEN GOTO 6040
6010  LET SP = SP + 1
6020  LET V = STACK (SP)
6030  RETURN
6040  PRINT "STACK EMPTY"
6050  STOP
```

EXAMPLE: THE TOWER OF HANOI

The "Tower of Hanoi" puzzle provides a simple example of the use of stacks. In its initial state, the puzzle looks like this:

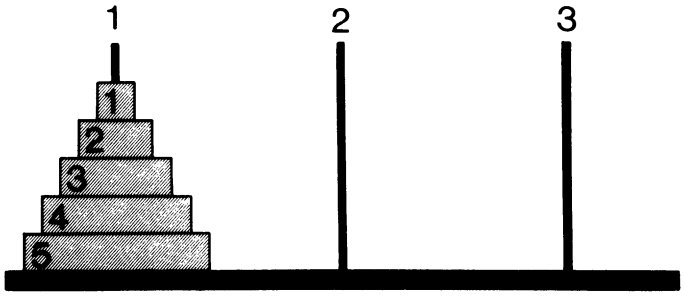


Figure 3.1

There are five discs with holes drilled through their centres, mounted on a spindle, with the largest disc at the bottom and the smallest at the top as shown in Figure 3.1. A disc may be removed from the top of this heap (or any other that forms) and placed on one of the other two spindles.

Obviously, we've got three stacks. But there's an extra constraint: at no stage may a larger disc sit on top of a smaller one. The goal is to transfer the whole tower to one of the other spindles.

Our problem is to allow a player to make moves, and display the state of the discs at each stage, checking for illegal moves.

Since there are three stacks we need to make some modifications. We'll have a pointer P to the current stack, so that line 10 becomes

```
10  DIM STACK (20, 3)
```

Also we need three separate stack pointers, which we'll put in an array called SP:

```
15  DIM SP (3)
20  LET SP (1) = 20
21  LET SP (2) = 20
22  LET SP (3) = 20
```

(Incidentally we're only using 20 here because it was used before: a smaller stack would be fine.) Now all references in PUSH and POP to STACK(SP) become references to STACK (SP (P), P). For instance, line 5010 reads:

```
5010  LET STACK (SP (P), P) = V
```

and 5020 becomes

```
5020  LET SP (P) = SP (P) - 1
```

Now, before we call PUSH or POP, we have to ensure that P is set to the right stack. Here goes. The first job is to set up the left-hand stack (P = 1) to contain the discs. We can identify these by the numbers used in Figure 3.1. So:

```
100  LET P = 1
110  FOR V = 5 TO 1 STEP - 1
120  GOSUB PUSH
130  NEXT V
```

Then we find out which stack the player wishes to remove a disc from:

```
140  PRINT "WHICH PILE TO BE REDUCED?"
150  INPUT SR
```

and we unstack that value

```
160  LET P = SR
170  GOSUB POP
```

At this stage the required value is in V. We don't need to test whether there was a disc in the stack to remove, because POP tests for an empty stack anyway! Now ask where the disc is to be put:

```
180  PRINT "WHICH FILE TO BE INCREASED?"
190  INPUT SI
```

Now see if the move is legal. This is going to need some thinking about, so we'll put off the day of reckoning and simply define a subroutine, from which a return only occurs if the move is valid:

```
200  GOSUB LEGALCHECK
```

So if 210 is reached the move was legal, and we can PUSH into the stack SI, and then repeat the whole moving experience . . .

```
210  LET P = SI
220  GOSUB PUSH
230  GOTO 140
```

It remains to attack LEGALCHECK. We need to confirm that the value on top of the SI stack is greater than the value in V. That involves POPping this stack—which will overwrite the value in V! So we'll save V first:

```
300  REM LEGALCHECK
310  LET VS = V
320  LET P = SI
330  GOSUB POP
340  IF V > VS THEN 370
350  PRINT "ILLEGAL"
360  STOP
370  GOSUB PUSH
380  LET V = VS
390  RETURN
```

Note lines 370 and 380: having popped the value off the stack to examine it we mustn't forget to push it back on, to restore its state. Also, since we moved V out into VS temporarily, we have to restore its original value, otherwise the PUSH at line 220 will be pushing the wrong thing!

And now I have to tell you that this won't work. The reason is that we haven't considered the special case that a stack is empty when something is added to it. Under these circumstances the program will never get past line 330, calling POP: it will simply object that it can't, because the stack is empty. The simplest thing would be to put "6" at the bottom of all three stacks to start with, so that any value in the range 1–5 is allowed to stack on top of it. I'll leave that (or the alternative: test if the stack is empty, and PUSH straight away) as an exercise for you.

I'll also leave you with the problem of *displaying* the stacks as the game progresses. This can be as simple as printing out the internal values, if you like; or more satisfyingly, plotting each disc, using the internal value 1–5 to define the length of the line of graphics characters used.

One final comment: I've adopted a new convention here of allowing *named subroutines* (GOSUB, PUSH, etc.) Obviously that's for improved readability; but it's quite legitimate as a program instruction, provided the variable (here PUSH) is given the value of the subroutine's starting line number. So for instance

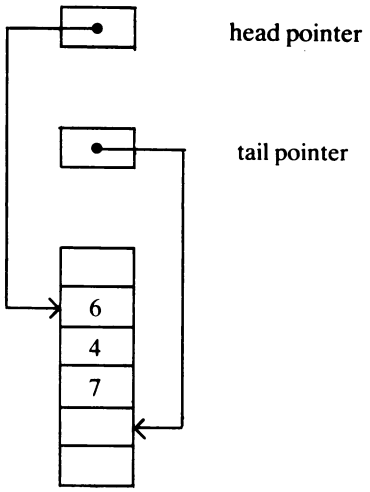
```
1  LET LEGALCHECK = 300
```

will allow GOSUB LEGALCHECK to be interpreted, correctly, as GOSUB 300. This idea can also save work if you decide to change the subroutine addresses.

How to store information temporarily and process it in the order in which it arrived.

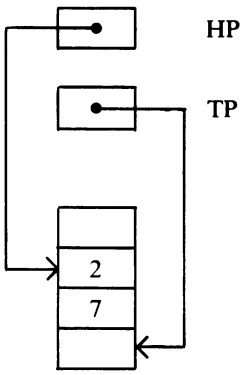
4 Queues

Again, the word "queue" describes the operation of the structure well enough. Items are added to the queue at one end, but removed from the other. So now we need two pointers, one for the head of the queue and the other for the tail, like this:

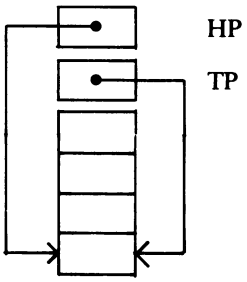


Implementing a queue is very like implementing a stack, but there's one extra problem, to do with testing the queue to see if it's full. For instance, suppose we add two items to the queue shown above, so that the tail pointer is now beyond the array. If we didn't think too carefully about it, we could use that information to decide that the queue is full. Of course, it isn't, because there's a spare slot above the head, and if we remove items from the queue there will be more available space there, although the tail pointer hasn't moved, and it's the tail pointer we're testing to see if the queue is full.

So let's rethink the problem. Firstly, when the tail pointer falls off the bottom of the array we want it to reappear at the top, so as to use the available array space. In other words, the array becomes a circular structure, and both pointers chase each other round it. That still leaves the question of how we tell when the queue is full or empty. Let's think about a simple example:



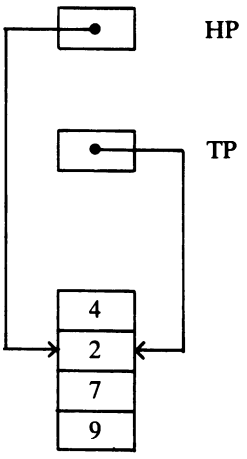
If we remove two items from this queue, we'll have to add two to the head pointer and the queue will now be empty:



(For clarity, I've removed the 2 and 7 from the array, although, as with the stack, they will actually be left there).

So there's a nice simple rule to determine when the queue is empty: the head and tail pointers are equal.

Now let's fill the original queue by adding two values (9 and 4, say) to it. So the tail pointer goes back to the beginning of the array and then gets incremented by one:



Oops! The head and tail pointers point to the same place again, so all our rule actually tells us is that the queue is either full or empty. Not much use really . . .

Never mind! We can get round it quite easily. What we've really done so far is roughly like taking a snapshot of a motor race and asking somebody to look at it and say which car is in the lead. Of course, he can't, because he doesn't have any information about how many laps each car has done. We're in a similar position. The pointers could be neck and neck (empty) or the tail pointer could be about to lap the head pointer (full). So we don't need to know exactly how many laps each pointer has done, only the difference between them. This can only be zero or one, because to allow more would overfill the array.

QUEUE ROUTINES

Having dealt with that problem, the rest of the programming is pretty similar to that for a stack. Again, we need three routines, called INITIALIZE, ENQUEUE and DE-QUEUE. As before, INITIALIZE can appear at the beginning of the program:

```
10  DIM QUEUE (20)
20  LET HP = 1
30  LET TP = 1
40  LET LAP = 0
```

the ENQUEUE routine is:

```
2000  IF HP = TP AND LAP = 1 THEN GOTO 2060
2010  LET QUEUE (TP) = V
2020  LET TP = TP + 1
2030  IF TP > 20 THEN LET LAP = LAP + 1
2040  IF TP > 20 THEN LET TP = 1
2050  RETURN
2060  PRINT "QUEUE FULL"
2070  STOP
```

A couple of things to note:

As for a stack, I'm assuming an array of length 20, and the value to be enqueued to be held in V. Line 2000 tests to see if the queue is full. Lines 2030 and 2040 deal with the problem of completing a lap. Since two actions are necessary to do this, we can either ask the same question twice (as I have) or let:

```
2030  IF TP > 20 THEN GOSUB 2080
```

perform the two actions on 2080 and 2090 and have another RETURN on 2100.

Logically they are equivalent, but if you spray too many GOSUBs and GOTOs around your code, it looks like undisciplined knitting, and becomes very difficult to read.

DEQUEUE takes the form:

```
3000  IF HP = TP AND LAP = 0 THEN GOTO 3060
3010  LET V = QUEUE (HP)
3020  LET HP = HP + 1
3030  IF HP > 20 THEN LET LAP = LAP - 1
3040  IF HP > 20 THEN LET HP = 1
3050  RETURN
```

```
3060 PRINT "QUEUE EMPTY"
3070 STOP
```

See how similar the two subroutines are? There's a kind of mirror-image symmetry about them. The same was true of the PUSH and POP routines. It's a very common phenomenon in routines which do, in some sense, opposite things. The programmer with a nose for trouble will worry when symmetry like this isn't present. It's not a hard and fast rule; just a hunch that comes with experience, but useful nevertheless.

A FOOTBALL QUEUE

Here's an example of how a queue might be used.

The ABC video printer, used on weekend sport programs to transmit the football results, is really just a kind of long-distance typewriter. And, often, it looks like it—with a one-finger merchant operating it from the far end. Suppose instead that, when the operator types a character, it is not immediately transmitted, but queued. The entire queue is dequeued only when the operator hits "return". That way "ARSENAL" will flash on to the screen as a whole word, instead of being laboriously typed A-R-S- . . .

Here's the necessary code:

```
10 LET EMPTY = 0
20 LET HP = 1
30 LET TP = 1
40 LET LAP = 0
110 LET V$ = INKEY$
120 IF V$ = "" THEN GOTO 110
130 IF V$ = CHR$ 118 THEN GOTO 160
140 GOSUB ENQUEUE
150 GOTO 110
160 FOR I = 1 TO 20
170 GOSUB DEQUEUE
180 IF EMPTY = 1 THEN GOTO 210
190 PRINT V$;
200 NEXT I
210 PRINT
220 LET EMPTY = 0
230 GOTO 110
```

There are some slight modifications needed to ENQUEUE and DEQUEUE, because we're dealing with characters, not numbers. In particular V becomes V$, and so on, and the arrays become character arrays. When the queue is flushed we don't want "QUEUE EMPTY" appearing on the screen—imagine the viewers' surprise at this little-known team playing against Chelsea—so we replace line 3060 of DEQUEUE by

```
3060 LET EMPTY = 1
```

Now we simply test EMPTY on the return to see if the queue is empty yet; and if it is, we start on the next word.

We are here using EMPTY as a *flag*: we wave it (i.e. set it to the value 1) to show that something interesting has happened. Then, to find out later if it did happen, we just look

at the flag. If it's at Ø, then it didn't happen after all. Flags are important later in the machine code chapters.

At the moment you'll get a printout like

```
ARSENAL
2
LIVERPOOL
3
```

because an ENTER is generated at every queue-flushing operation by line 21Ø. (In fact, if you're too slow on the keys, you'll get AAARRRSSSSSEENNNNAAAALLLL, with auto-repeat!) Also, since you can't input SPACE with INKEY$—it would be interpreted as BREAK— you'll also get

```
WEST
HAM
4
QUEENS
PARK
RANGERS
1
```

See if you can devise ways round these faults, to give output like

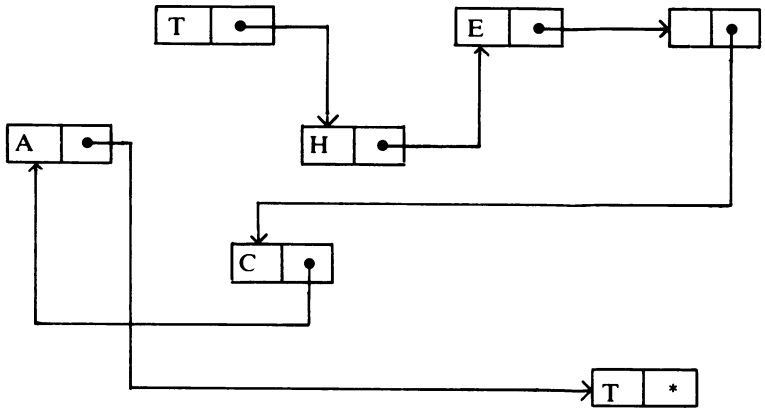
```
WEST HAM  4    QUEENS PARK RANGERS  1
```

For another example of the use of queues see *Checkout*, Chapter 9.

The treasure-hunt principle for easily modified listings.

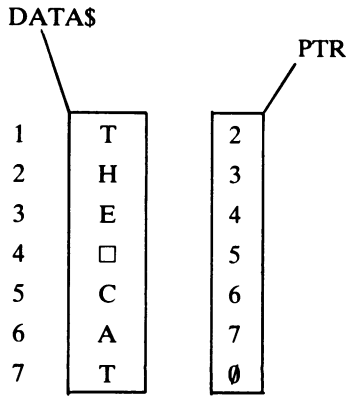
5 Linked Lists

The structures we've looked at so far have all had one thing in common; if you know where a particular element is, you know where the next one is, because it's always in an adjacent memory cell. But suppose we allowed related data to be sprinkled all over memory? (Let's not worry for the moment about why we might want to.) Then we would need some explicit way of finding an item when we know where the preceding one was. The diagram below illustrates this in an abstract way.



The symbols T, E, A, H, C and "space" are arranged in no obvious pattern until you take into account that each is associated with a pointer so that provided you start at the T on the top line and follow the pointers through to the asterisk, which I'm using as a delimiter to mean "end of list", you get the message:"THE CAT". It's the treasure-hunt principle, isn't it? Find an item, and that gives you a clue to the next.

How do we implement the structure? Well, each element has two components, some data and a pointer. So why not have a pair of corresponding one-dimensional arrays organized as shown overpage:



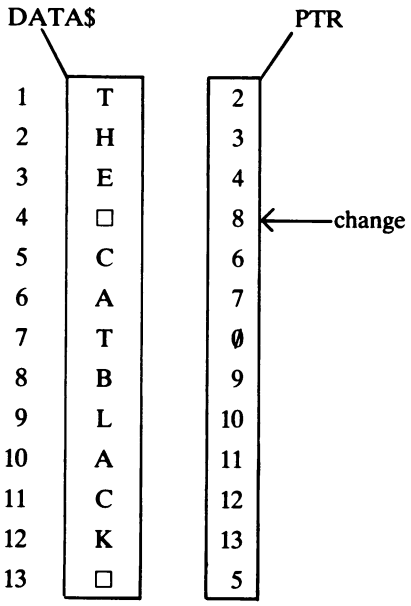
so that if we want to print the contents of DATA$ in the right order we need a subroutine like:

```
1000  LET P = 1
1010  PRINT DATA$ (P);
1020  IF PTR (P) = Ø THEN RETURN
1030  LET P = PTR (P)
1040  GOTO 1010
```

(Note that I've had to use zero, rather than asterisk, for my delimiting pointer because PTR is a number array.)

"Hang on a minute!" You're all objecting (like mad by now I expect). "You don't need to make all this fuss. You could just print out the contents of DATA$ in a FOR loop from 1 to 7."

That's right. But suppose I want to alter the message to "THE BLACK CAT". Doing it the straightforward way, I'd have to move the letters C A T down the array to make room to insert BLACK. Using the linked list, all I need to do is to tag BLACK on to the end of the array and change one of the original pointers (5) like this:



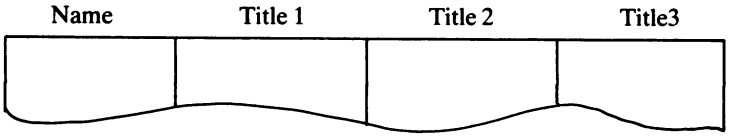
OK, it wouldn't have been a great problem to move three letters five places down the array, but suppose these were the first words of a five thousand word essay?

What we've just done is to edit a piece of text. That's the main job of what have come to be called "word-processing" programs, and I've just introduced the fundamental data structure on which most word-processors depend. I don't want to take this example any further just now, because the word-processor case study (Chapter 8) deals with it in some depth.

A LIBRARY CATALOGUE

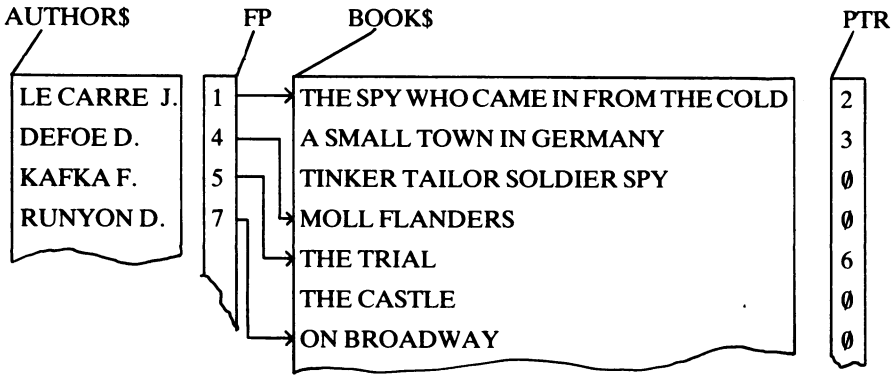
However, the linked list is such a useful structure that it's worth looking at another application for it. Let's suppose that a librarian wants to keep an author index so that when a subscriber asks "have you got any other books by Austin P. Goatwhirler?" he can answer immediately, even though he is not a Goatwhirler fan himself.

We might, first of all, hit on the idea of using a simple table like this:



but, of course, the disadvantage with that is that the number of books which can be recorded for each author is fixed, and some authors may only have one book in the library, in which case a lot of memory is being wasted, while others may have more books than we can allocate space for.

An organization in which each author has his own linked list of book titles gets over this problem. There'll have to be an author array with a corresponding pointer array to point to the first entries in the book lists like this:



To list all the books by a particular author, all we need to do is match the author's name with the corresponding first pointer:

```
500 PRINT "ENTER AUTHOR"
510 INPUT A$
520 FOR P = 1 TO 200                    [if there are 200 authors]
530 IF A$ = AUTHOR$ (P) THEN GOTO 570
540 NEXT P
550 PRINT "THERE IS NO REFERENCE TO THIS AUTHOR"
560 RETURN
```

When we get to line 570, P points to the target author, and also to the corresponding position in the FP array. So that FP (P) tells us where to start looking in BOOK$.

So the rest of the routine looks very like the previous linked list printout program:

```
570  LET P = FP (P)
580  PRINT BOOK$ (P)
590  IF PTR (P) = 0 THEN RETURN
600  LET P = PTR (P)
610  GOTO 580
```

All this assumes that the AUTHOR$, FP, BOOK$ and PTR arrays have already been correctly set up. Our librarian will not thank us if he has to know about the internal structure of his data in the primitive way that we've just thought about it. So we need another routine which allows him to enter the data in a more natural way, and sets up the links that we need automatically. I'll leave you to think about that problem. It's quite an interesting little project. Just to give you some pointers (pun!) to it, I'll consider a related routine, one to make an insertion of a new book into an existing author catalogue.

To take a particular example, suppose we add Franz Kafka's *America* to the library. First, we search the BOOK$ array for the first unused element, and insert "AMERICA" into it, keeping a note of where it is (element 8, in our example). At the same time we can set the corresponding element of PTR to zero, since the new book is now the last entry under this author. So PTR (8) = 0 in this case. The only other thing to do is replace the old zero delimiter by 8, so that "AMERICA" will be printed after "THE CASTLE" in the print routine. Of course, it's no good just searching the PTR array for a zero, because we need the one which terminates Kafka's novels. So we have to look for KAFKA F. in AUTHOR$, find 5 in the corresponding FP element, look in PTR (5), find 6, look in PTR (6), find zero and replace it by 8.

The routine is:

```
1500  PRINT "ENTER AUTHOR"
1510  INPUT A$
1520  PRINT "ENTER NEW TITLE"
1530  INPUT NT$
1540  FOR P = 1 TO 1000                    [if BOOK$ is 1000 elements long]
1550  IF BOOK$ (P) = " "THEN GOTO 1570      ┐  search for
1560  NEXT P                                ┘  1st null entry
1570  LET BOOK$ (P) = NT$                   ┐  insert new
1580  LET PTR (P) = 0                       ┘  title
1590  FOR I = 1 TO 200                      ┐
1600  IF A$ = AUTHOR$ (I) THEN GOTO 1620    ┘  search for author
1610  NEXT I                                ┐
1620  IF PTR (I) = 0 THEN GOTO 1650         ┘  search for terminating zero
1630  LET I = PTR (I)                       ┐
1640  GOTO 1620                             ┘
1650  LET PTR (I) = P                       ┘  replace it with P
1660  RETURN
```

MENUS

We've got the basis, here, of a genuinely useful data-retrieval system. Obviously, there are a lot more routines which would be necessary (for instance we don't have a way of deleting an entry at the moment), and we need a way of linking them together. A convenient technique is to use a menu. When the program is run, it first displays on the screen a list of options which it can execute. So, in this example, we might get something like:

```
LIBRARY RETRIEVAL SYSTEM
------------------------

OPTIONS ARE:

  1. SET UP NEW LIBRARY
  2. INSERT
  3. DELETE
  4. SEARCH

ENTER OPTION (1–4):
```

When you enter one of the options you may get a sub-menu. In this case if 2 is entered, you could get:

```
INSERT ROUTINE
--------------

OPTIONS ARE:

  1. AUTHOR INSERT
  2. NEW TITLE

ENTER OPTION (1/2):
```

This way, all the routines can be written as independent subroutines, the menus can just be printed out at the beginning, and calling the right subroutine can be achieved with:

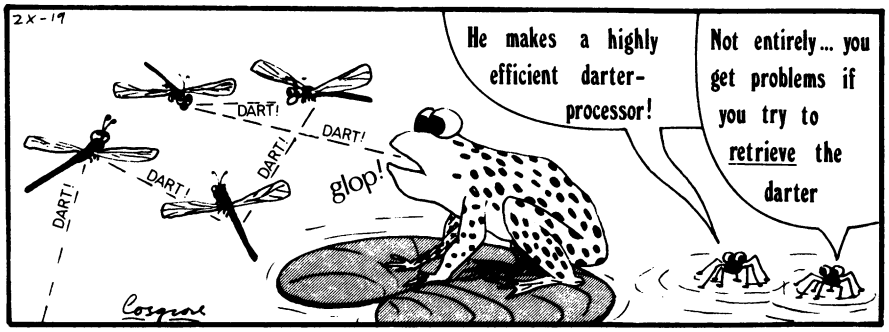
```
50  GOSUB 1000 * OPT
```

where OPT is the value input at the bottom of the menu. We just have to ensure that the SET UP NEW LIBRARY routine is at 1000, INSERT is at 2000, DELETE is at 3000 and so on. The INSERT routine would have something similar:

```
1020  GOSUB (1000 + OPT * 300)
```

The AUTHOR INSERT routine would have to be at 1300 and the NEW TITLE routine at 1600.

There's more about breaking programs down into manageable chunks in the section on *Structured Programming*.



There are plenty of other features which you can add to this embryo data retrieval system. How about a subject-index, for example? A SUBJECT$ array will be necessary (with its associated FP array), behaving just like the AUTHOR$ array. There's no point in repeating the information in BOOK$, but the linking pointers will be different, so we need a new PTR array.

At the moment, it is not easy to find out who wrote a particular book. Why not have a set of pointers pointing from BOOK$ into AUTHOR$ to handle that?

And so on, and so on. But I'll hand over to you, at this point. There's a lot of interesting programming here, and a really useful program at the end of it.

Project

I've alluded several times in this section to problems in Stock Control. It isn't particularly difficult to write a suite of programs to handle the stock for a small business.

You'd start by considering what kinds of data are required, and a suitable structure for these. You'd certainly need things like part number, part description, unit cost, number in stock, location (bin number), reorder level, reorder number, supplier's address. The chances are that arrays organized as tables will do for this, unless there are alternative suppliers of some items, in which case linked lists would be convenient.

Then you'd need routines like these:

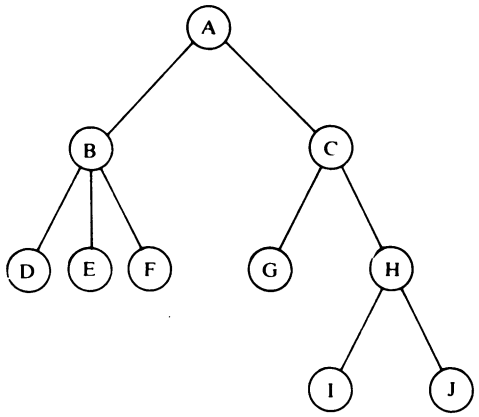
1. Add new stock.
2. Remove stock.
3. Interrogate system for (a) number in stock of a given item,
   (b) location of a given item.
4. Add new item.
5. Delete item.
6. Change supplier of item.
7. Change price of item.
8. Generate orders for items at below reorder level.
9. Generate financial reports (such as average cash tied up in stock).

Like all "real" projects this can grow like Topsy. The important thing is to make sure that each routine is independent of the others so that new ones can be added, and old ones can be edited, with the minimum of bother.

An old genealogist's trick,
for tasks which branch at every stage.

6 Trees

A *tree* is a structure consisting of *nodes* and *branches*. Each node (except one) has exactly one branch entering it, and may have any number, including zero, leaving it. The exception is the root, which has no branches entering it. A node which has no branches leaving it is called a *leaf*. So, a tree might look like:



The lettered circles are all nodes, joined by straight line branches. "A" is the root and "D", "E", "F", "G", "I" and "J" are all leaves. Implementing this structure isn't much different from implementing a linked list, except that now there is a variable number of pointers from each node. In this case there are never more than three, so a three-column array will do:

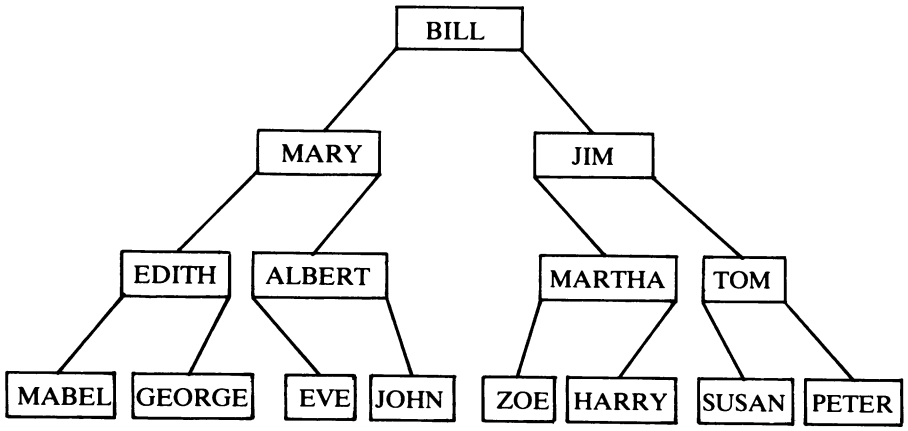
	DATA$	PTR		
1	A	2	3	Ø
2	B	4	5	6
3	C	7	8	Ø
4	D	Ø	Ø	Ø
5	E	Ø	Ø	Ø
6	F	Ø	Ø	Ø
7	G	Ø	Ø	Ø
8	H	9	10	Ø
9	I	Ø	Ø	Ø
10	J	Ø	Ø	Ø

A FAMILY TREE

There are some very straightforward applications for trees, and some less obvious ones.

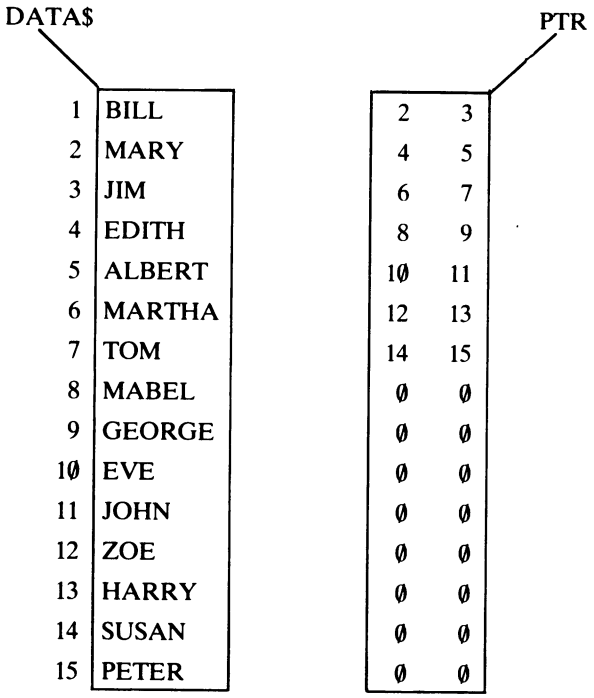
Let's look at a straightforward one first—the representation of a family tree.

This is another data retrieval problem, really. We store the family tree, and then want answers to questions like "who was X's maternal grandfather?" So we might consider an organization like:



In other words, Jim and Mary are Bill's parents, Albert and Edith are Mary's and so on. Of course, it's an incomplete representation, because there is, for example, no way of telling whether Albert and Edith had more than one child. It's really just that part of the complete tree which affects Bill directly. To get more detail, we would need pointers from each node to other trees showing the offspring of that node's brothers and sisters; which would lead to other trees in the same way. Boggle, boggle! Let's keep it simple.

The internal structure looks like:



	DATA$	PTR	
1	BILL	2	3
2	MARY	4	5
3	JIM	6	7
4	EDITH	8	9
5	ALBERT	10	11
6	MARTHA	12	13
7	TOM	14	15
8	MABEL	0	0
9	GEORGE	0	0
10	EVE	0	0
11	JOHN	0	0
12	ZOE	0	0
13	HARRY	0	0
14	SUSAN	0	0
15	PETER	0	0

SETTING UP A TREE

Genealogists are, in my experience, no more sympathetic to the problems of computer programmers than librarians are, so we'll need a way to set up this structure. Suppose we ask the user to enter the name of a member of the tree, together with his (or her) mother and father, in that order. We don't ask any more than this; that is, we don't insist that BILL is the first entry, for instance.

So we have:

```
100  PRINT "ENTER NAME"
110  INPUT N$ (1)
120  PRINT "ENTER MOTHER"
130  INPUT N$ (2)
140  PRINT "ENTER FATHER"
150  INPUT N$ (3)
```

Now we insert N$ (1), N$ (2) and N$ (3) into the DATA$ array. Anywhere will do, because the pointers are going to take care of the links, so the simplest thing is just to load them into the first three available spaces. Rather than have to read DATA$ every time this is done to find some free space, we could keep a pointer to the first free element. This will be 1 to begin with so:

```
90  LET PFF = 1        [pointer to first free]
```

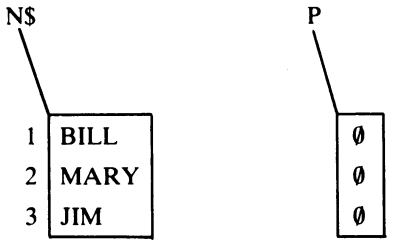
Hang on though! What if one (or even all) of these names have been entered before? We don't want double entries, so we'll have to search the array for each name in turn:

```
160  FOR I = 1 TO 3
170  LET P (I) = 0
180  FOR R = 1 TO 15
190  IF DATA$ (R) = N$ (I) THEN LET P (I) = R
200  NEXT R
210  NEXT I
```

It may have been worrying you why I bothered to enter the names into a new array (N$). Can you see that it's saved repeating lines 170 to 200 for three sets of values?

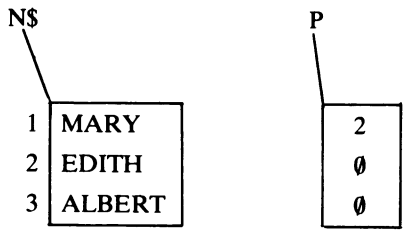
What's the inside loop doing? Well, if the name isn't already in DATA$, P (I) is left at zero. Otherwise, P (I) contains the row where the name is to be found.

Let's follow this through so far with the example. We enter BILL, MARY, JIM. The program searches for these names and doesn't find them so we're left with:



We can use this to make the rule: "If an element of P contains zero, the corresponding element of N$ can be entered into DATA$, because it hasn't occurred before".

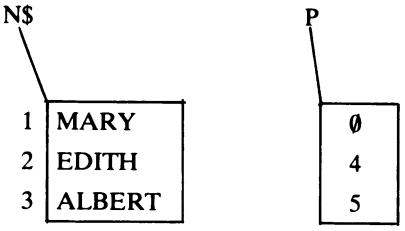
Now we enter MARY, EDITH, ALBERT. This time N$ and P appear as:



So only EDITH and ALBERT need to be copied. As we copy them, let's keep a record of where we put them by holding the row values in the P array:

```
220  FOR I = 1 TO 3
230  IF P (I) <> 0 THEN GOTO 270
240  LET DATA$ (PFF) = N$ (I)
250  LET P (I) = PFF
260  LET PFF = PFF + 1
270  NEXT I
```

So now we've got:



and that tells us that the pointers in row 2 should be 4 and 5, or in general, row P (1) contains the pointers P (2) and P (3)!

All we need to write is:

```
280  LET PTR (P (1), 1) = P (2)
290  LET PTR (P (1), 2) = P (3)
```

That's broken the back of the problem; now for some tidying up. First we need to loop what we've got, to allow all the names to be entered:

```
300  GOTO 100
```

and that means we need a way out of the loop:

```
115  IF N$ (1) = "*" THEN RETURN        [or GOTO wherever]
```

Finally, we've got to dimension all the arrays:

```
10  DIM DATA$ (15, 10)        [if no name exceeds 10 letters]
20  DIM PTR (15, 2)
30  DIM N$ (3, 10)
40  DIM P (3)
```

One more thing you should notice: since leaves don't have pointers leaving them, the initial values at leaves in the PTR array are never altered. Since these will be set to zero by BASIC when the array is dimensioned, and we're using zero to indicate "no pointer", this is fine unless, in the middle of the program, you want to grow a new tree in the same array. Then, you would have to zero the PTR array to leave it in the state the "Grow a tree" routine expects to find it. It would be safest to do this anyway at the beginning of the routine.

Now we come to a very interesting feature of the program we've just written. We have assumed throughout that the data are entered in the logical order—first BILL, MARY and JIM, then EDITH and ALBERT followed by JIM, MARTHA and TOM etc. but actually the order in which the sets of three names are entered doesn't matter a bit. Try it. Key in the program, tack a routine on the end to point to the DATA$ and PTR arrays and then enter, for instance, ALBERT, EVE, JOHN, then JIM, MARTHA, TOM, then MARTHA, ZOE, HARRY and so on. Of course, the positions of the names in the DATA$ array will vary depending on your order of entry, but the pointers will also change so that they point to the right names.

This is nice, because it allows the user to dredge bits of information out of his memory in no special order (which is how most of our memories work) and enter them while he remembers them. He doesn't have to form a notion of the tree that we've been working from.

Searching the tree for details of a person's parentage is easy. Find the required names in DATA$. The pointer in column 1 of PTR of that row points to his (or her) mother and that in column 2 points to the father. The process can be repeated to look for grandparents, great grandparents and so on. I'll leave the actual coding to you.

A common use for trees: storing moves in a game. Backtrack through the tree to find a winning strategy. Here the computer teaches itself to win at NIM.

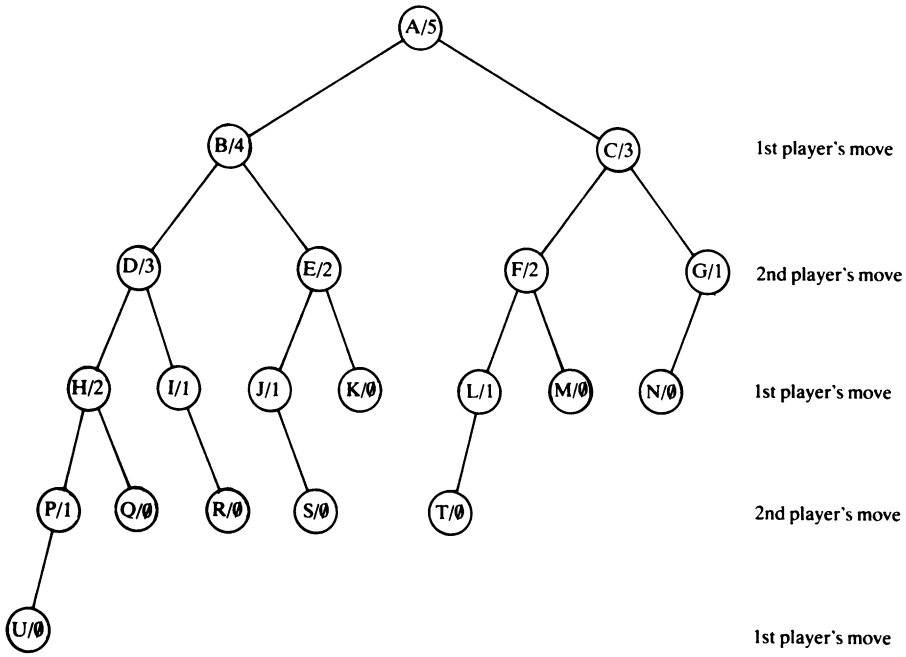
7 Game Trees

It's possible to represent the moves in a two-person game in a tree. Let's look at a simple example. I'll define the rules of a game as follows.

1. There are two players who take it in turns to move.
2. The initial state of the game is that five matchsticks lie on a table.
3. A legal move consists of removing either one or two matchsticks.
4. A player has won if the other player removes the last match.

You've probably recognized this as a simplified form of Nim (in which there are more matchsticks and more options) and if you've ever played Nim you'll know it's a very simple game, so what we've got is very simple indeed!

The tree shows every position that is possible in the game, and links are shown between successive possible positions. The letters in the nodes are just references, but the numbers indicate the number of matchsticks left at that stage:



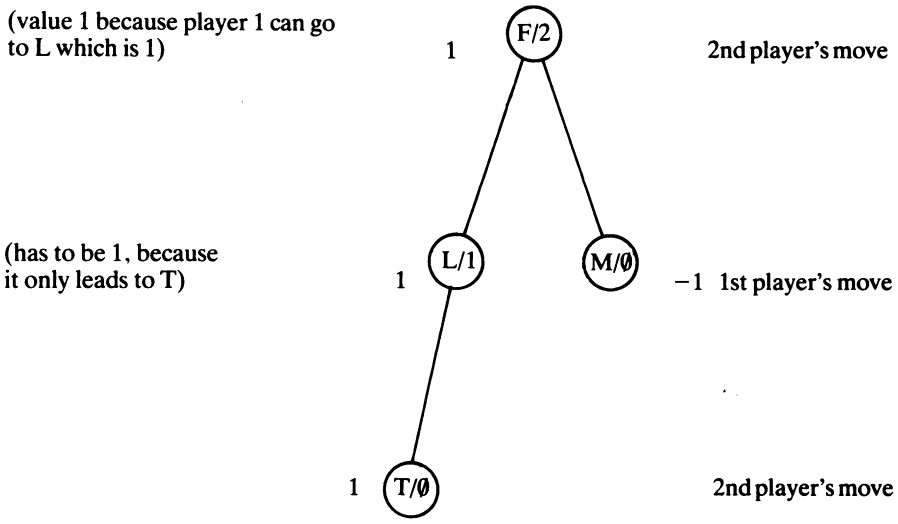
Notice that I haven't said anything yet about whether a move is good or bad. For instance, if the game reaches node E, player 1 certainly isn't going to remove two matches and so reach node K because in doing so he loses! But it is a legal move and the tree is only concerned with all possible legal moves.

WHEN IS A MOVE GOOD?

Now let's assign some values to the nodes which indicate the quality of a particular move. Obviously, this is easiest at the leaves because we know who's won. Let's define a value 1 to mean player 1 wins and a value −1 to mean player 2 wins. (I've used −1 out of a feeling of symmetry: you *could* try ∅.)

For instance, node U has the value −1 because player 1 picks up the last match. Similarly Q, R, S and T are all set to 1, and K, M and N to −1. In some cases, it's possible to assign values to nodes which aren't leaves. For example, since node P only branches to node U it must have the same value as node U (i.e. −1). This is another way of saying that once the game has reached node P, player 2 is guaranteed to win.

Now we'll make the assumption that each player plays his best move at each stage. That means that if player 1 has the option of going to a node whose value is 1 he'll take it, and so the node from which he's moving can be said to have the value 1. Similarly player 2 will try to make a node −1. Using this rule, we can "back up" the tree from the leaves evaluating each node as we go. Look at a small portion of the tree:



If you back up this tree to the root, you'll find the root evaluates to 1. In other words the first player can always win!

What this shows is that Nim isn't a very interesting game. Indeed, no game of which the complete set of moves can be written down is very interesting, because the implication is that it can be played in a totally automatic way, without any guesswork or inspiration. On the other hand, it's exactly that feature which should make it easy for a computer to play!

A PROGRAM FOR NIM

So, how would we go about designing a Nim playing program? Firstly, it's obvious that we need to store the tree:

		NM	NV	PTR	
1	A	5	1	2	3
2	B	4	1	4	5
3	C	3	−1	6	7
4	D	3	1	8	9
5	E	2	1	10	11
6	F	2	1	12	13
7	G	1	−1	14	0
8	H	2	−1	15	16
9	I	1	1	17	0
10	J	1	1	18	0
11	K	0	−1	0	0
12	L	1	1	19	0
13	M	0	−1	0	0
14	N	0	−1	0	0
15	P	1	−1	20	0
16	Q	0	1	0	0
17	R	0	1	0	0
18	S	0	1	0	0
19	T	0	1	0	0
20	U	0	−1	0	0
		node reference	number of matches	node value	left pointer	right pointer

We could set this up with a series of input statements inside a loop, or we could "grow" the tree in a similar way to that in which we grew the family tree. Either way, we'll assume that it's available in the form shown.

From here on, the problem's easy. At each stage, the computer has two possible moves, given by the PTR values in the current row. It simply chooses one which leads to a "1" in the node value array. So to start with, we have to initiate a row pointer to the root of the tree, display the number of matches and inform the human player that it's the computer's go (after all, we want the computer to win, don't we?):

```
1000 LET R = 1
1010 PRINT "THERE ARE □"; NM (R); "□ MATCHES ON THE TABLE."
1020 PRINT "IT'S MY GO!"
```

Notice that I've written NM (R) in line 1Ø1Ø rather than just "5" because this will enable us to use the same statement next time around.

Now to calculate the computer's move:

```
1Ø3Ø  FOR C = 1 TO 2
1Ø4Ø  IF NV (PTR (R, C) ) = 1 THEN GOTO 1Ø6Ø
1Ø5Ø  NEXT C
1Ø6Ø  LET R = PTR (R, C)
1Ø7Ø  PRINT "THERE ARE □"; NM (R); "□ MATCHES ON THE TABLE."
```

The loop (1Ø3Ø–1Ø5Ø) looks for a pointer in the current row which points to an NV value of 1. When it finds one it updates the current row to this value. There's no real need for a loop, since there are only two columns (i.e. two possible moves) to consider. But using one means that this form of routine will work for more complicated games where there are dozens of possible moves and correspondingly many columns. We only have to change the "2" in line 1Ø3Ø to however many columns there are. Note also that we only need to find the first "1" in NV. After all, any "1" leads to a win.

Now we allow the human player a turn:

```
1Ø8Ø  PRINT "YOUR TURN NOW"
1Ø9Ø  PRINT "HOW MANY MATCHES DO YOU WANT TO PICK UP?"
11ØØ  INPUT HM
111Ø  IF HM > Ø AND HM < 3 THEN GOTO 114Ø
112Ø  PRINT "NO CHEATING"
113Ø  GOTO 1Ø9Ø
114Ø  LET R = PTR (R, HM)
```

Now to see if we've won yet, and if not, to play again:

```
115Ø  IF NM (R) > Ø THEN GOTO 1Ø1Ø
116Ø  PRINT "I'VE WON AGAIN!"
```

If we want to be generous, we could allow the user to go first every other game, by passing control to the "computer move" and "human move" routines in the opposite order. That would mean including lines in the move routine to see if the human player has won, and treating the two "play" routines as separate subroutines rather than in-line code. If there's a variable called G which counts the number of games played, and the two routines begin at 1Ø2Ø and 1Ø8Ø, as they do here, the calling program could look something like:

```
5ØØ  LET G = Ø
51Ø  GOSUB (1Ø2Ø + 12Ø * (G/2 − INT (G/2) ) )
52Ø  GOSUB (1Ø8Ø − 12Ø * (G/2 − INT (G/2) ) )
53Ø  LET G = G + 1
54Ø  GOTO 51Ø
```

(don't forget to terminate the two "play" routines with RETURNs).

There are various other "cosmetic" modifications to be made (at the moment it displays: THERE ARE 1 MATCHES ON THE TABLE, for example), but I'm more concerned with another problem:

MAKING THE COMPUTER DO THE WORK

So far, we've assumed that the arrays have to be set up manually, in one way or another. But do they? After all, we set up the tree, and hence the arrays, from a knowledge of the rules of the game, and nothing else. Couldn't we give the computer the rules and get it to generate the array values in a similar way?

Let's give it a whirl. The NM array shouldn't be difficult; we just start with 5 in row 1 and use the rules to subtract 1 or 2 from this at each stage. The PTR array will have to link these values, and so far we have (without ever actually saying so) adopted the convention that the rule "Remove one match" is handled by the first column (rule 1) and the rule "Remove two matches" is handled by the second (rule 2). So we'll stick to that. Of course, the NV array will have to wait until we've grown the tree, because we need to "back up" to get the node values.

Assuming the arrays are already dimensioned, we can start with:

```
90  LET R = 1                          [first row]
100 LET NM (1) = 5                     [set the root]
110 LET CP = 2                         [set the "current pointer" to the first
                                        free row]
120 LET NM (CP) = NM (R) - 1           [rule 1]
130 LET PTR (R, 1) = CP                [link the pointer for rule 1]
140 LET CP = CP + 1                    [update 1st free row]
150 LET NM (CP) = NM (R) - 2           [rule 2]
160 LET PTR (R, 2) = CP                [link the pointer for rule 2]
170 LET CP = CP + 1                    [update 1st free row]
180 LET R = R + 1                      [go to next node]
```

Now we would like to go back to line 120, of course, to deal with the next node, but we need to avoid getting into an endless loop. We could cheat by remembering that there are twenty rows in the arrays, and branch out of the loop when we've dealt with all twenty; but bear in mind that we're trying to use this game to develop techniques which will still be applicable in more complex situations, when probably we won't know how many nodes there are in the tree.

Actually, there's an implicit rule we haven't used yet; namely, that you can't pick up a match which isn't there. So when matches are subtracted at lines 120 and 150, we ought to test to see that NM (CP) does not go negative. If it becomes zero, that implies that we've reached a leaf. Maybe we can use this to get out of the loop. If we insert:

```
115 IF NM (R) - 1 < 0 THEN GOTO 180
145 IF NM (R) - 2 < 0 THEN GOTO 180
```

the "negative match" problem goes away.

Now let's think about the way the pointers, R and CP, behave. CP hares off at twice the rate of R to begin with, since it's updated twice every loop to R's once. When the leaves start to be reached, CP slows down and eventually stops at 20, because the tests at lines 115 and 145 cause jumps around the "CP update" lines. All this time, R is plodding along, tortoise-like, at one update per loop. When it catches up with CP, we know there have been no forward entries past this row. So all we need is to test if R = CP yet:

```
190 IF R < CP THEN GOTO 115
```

Now we're left with generating the NV array values. Obviously we have to start at the leaves. We can find them easily enough, since the NM value at a leaf is zero. But in order to give the node a value we have to know whose turn it was.

Let's set up another array called WPM (which player to move) whose values will be set to 1 or 2 according to whether player 1 or player 2 is about to move. Here it is, added to the previous table.

		NM	NV	PTR		WPM
1	A	5	1	2	3	1
2	B	4	1	4	5	2
3	C	3	−1	6	7	2
4	D	3	1	8	9	1
5	E	2	1	10	11	1
6	F	2	1	12	13	1
7	G	1	−1	14	0	1
8	H	2	−1	15	16	2
9	I	1	1	17	0	2
10	J	1	1	18	0	2
11	K	0	−1	0	0	2
12	L	1	1	19	0	2
13	M	0	−1	0	0	2
14	N	0	−1	0	0	2
15	P	1	−1	20	0	1
16	Q	0	1	0	0	1
17	R	0	1	0	0	1
18	S	0	1	0	0	1
19	T	0	1	0	0	1
20	U	0	−1	0	0	2
node reference		number of matches	node value	left pointer	right pointer	which player to move

So in row 1, the WPM value is 1. Now, the rest of the WPM array is a bit puzzling, because there are varying numbers of 1s and 2s in each block. In an ideal world they would simply double each time: 1, 2, 4, 8 and so on, because there are two branches leaving every node. They don't of course, because some branches would lead to illegal moves. So we need a limit on the number of rows we examine with each loop. Initially, the first and last entries in WPM are at row 1 so:

```
210 LET FST = 1
220 LET LST = 1
```

To start with, player 1 is making a move so:

```
230 LET PLAYER = 1
```

Now we want to change the player:

```
240  LET PLAYER = (2 AND (PLAYER = 1) ) + (1 AND (PLAYER = 2) )
```

That's a sneaky piece of code which will swap PLAYER from 1 to 2 and vice versa. It works because the value of a condition is 1 if the condition is true and zero otherwise. So one of the parts of the expression disappears and the other is, if you like, multiplied by 1. Now, we'll loop from FST to LST counting the entries in WPM as we go:

```
250  LET ENTRIES = 0
260  FOR R = FST TO LST
270  IF PTR (R, 1) = 0 THEN GOTO 330     [this branch doesn't lead anywhere!]
280  LET WPM (PTR (R, 1) ) = PLAYER
290  LET ENTRIES = ENTRIES + 1
300  IF PTR (R, 2) = 0 THEN GOTO 330     [neither does this one]
310  LET WPM (PTR (R, 2) ) = PLAYER
320  LET ENTRIES = ENTRIES + 1
330  NEXT R
```

Now look at the next block:

```
340  LET FST = LST + 1
350  LET LST = FST + ENTRIES - 1
```

and see if we're over the end of the array yet:

```
360  IF FST < CP THEN GOTO 240
```

EVALUATING THE MOVES

Now we can set the NV values for the leaves:

```
370  FOR R = 1 TO 20
380  IF NM (R) > 0 THEN GOTO 410
390  IF WPM (R) = 1 THEN LET NV (R) = 1          [here only for a leaf]
400  IF WPM (R) = 2 THEN LET NV (R) = - 1
410  NEXT R
```

Next we search backwards through the tree, linking the node values. There are three situations to consider:

1. There's only one branch, so we can just pass the value back.
2. There are two branches, and the search finds the left link.
3. There are two branches, and the search finds the right link.

In the last two cases, the code is going to be several lines long, testing which pointer points to the maximum node value, and whether we want to pass back the minimum or maximum value, depending on who is about to play. So we'll put it in a subroutine starting at line 8000. The code is:

```
420  FOR R = 20 TO 2 STEP - 1
430  FOR RP = 1 TO 20
```

```
440  IF PTR (RP, 1) = R AND PTR (R, 2) = 0 THEN NV (RP) = NV (R)
450  IF PTR (RP, 1) = R AND PTR (RP, 2) < > 0 THEN GOSUB 8000
460  IF PTR (RP, 2) = R THEN GOSUB 8000
470  NEXT RP
480  NEXT R
```

Notice that, to make the code simple, all twenty rows are always searched in the inside loop. Of course, they don't need to be, because there's only one reference to a given pointer in the PTR array. So the code is inefficient, but tidy.

Now for the subroutine: let's make a table showing all the possible combinations of node values being pointed to, and show also what each player would want as a backed-up node value:

Left pointer node value	Right pointer node value	Player 1	Player 2
−1	−1	−1	−1
−1	1	1	−1
1	−1	1	−1
1	1	1	1

In the first and last cases, there is no choice in the backed up value, since both branches lead to the same number.

What we ought to do is determine in each case what the maximum and minimum value is, but this seems a bit like overkill for just two branches. (If there were more, we would have to do this.) Suppose we add the two node values in each line:

```
−2
 0
 0
 2
```

Now, only when this value is −2 is a "−1" forced to back up and only when it's 2 is a "1" forced to back up:

```
8000  LET TV = NV (PTR (RP, 1) )+ NV
      (PTR (RP, 2) )                    [add the node values]
8010  IF WPM (RP) = 2 THEN GOTO 8050
8020  LET NV (RP) = 1                   [for player 1 assume node is 1]
8030  IF TV < 0 THEN LET NV (RP) = −1   [if it can't be, set it to −1]
8040  RETURN
8050  LET NV (RP) = −1                  [for player 2, assume node is −1]
8060  IF TV > 0 THEN LET NV (RP) = 1    [if it can't be, set it to 1]
8070  RETURN
```

OTHER GAMES?

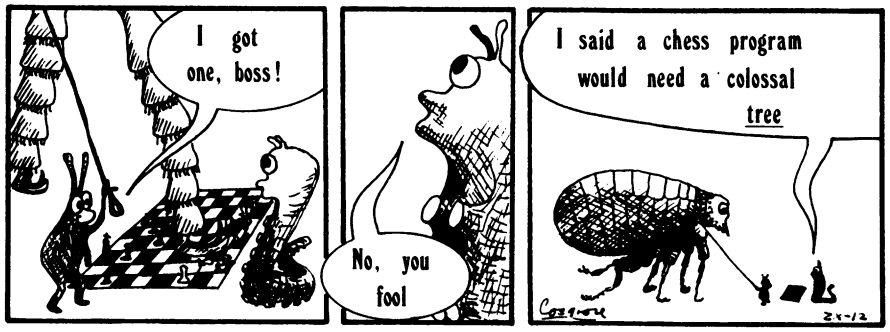
And that's it! A computer program which works out its own strategy! OK, the code is less than elegant in places; and here and there I've made use of features of this particular game, rather than making things totally general; but I was concerned with making it as easy as possible to follow the code. (Anyway that's my story.) It shouldn't be difficult to apply these principles to the design of a noughts-and-crosses-playing-program, for instance. (Remember though, that in this case it's possible for the game to be drawn so node values of zero will appear.)

More complicated games have a serious problem, however. The trees get colossal. Take chess for example. The first player has a choice of 20 moves to start with. (Two moves with each of eight pawns and two moves with each knight). The second player has exactly the same options from each of the 20 nodes just generated, so there are 421 nodes (including the root) after just one move each!

The solution is not to grow the whole tree—it simply wouldn't be possible, however large a machine you had—but to restrict it to, say, five moves ahead. Of course, that means it has to be regrown every move, and it also means that there may not be any leaves in the region we're looking at. That's a problem, because our whole technique has depended on setting the leaf values and backing up the tree.

What we have to do is to give values to the terminal nodes in the portion of the tree we have stored, according to some set of rules. At the simplest level we score high if we win material and low if we lose pieces. Then we add some "fine tuning". For example, deduct points if our king is unprotected, add them on if our queen controls the centre files. How many points to allot for a particular feature is pretty much a matter of intelligent guesswork to start with. Scoring will become modified with experience of how well, or badly, the program plays.

I don't recommend that you leap into action and write a chess-playing program, but there are a number of simpler games which are relatively easy to program and at which your ***1000*** can be made to play a surprisingly mean game. Connect-4 and Othello spring to mind. But BASIC versions are pretty *slow*: read the *Machine Code* section if you want to speed up the action—and then prepare for a long job of programming!



TREES AND INTELLIGENCE

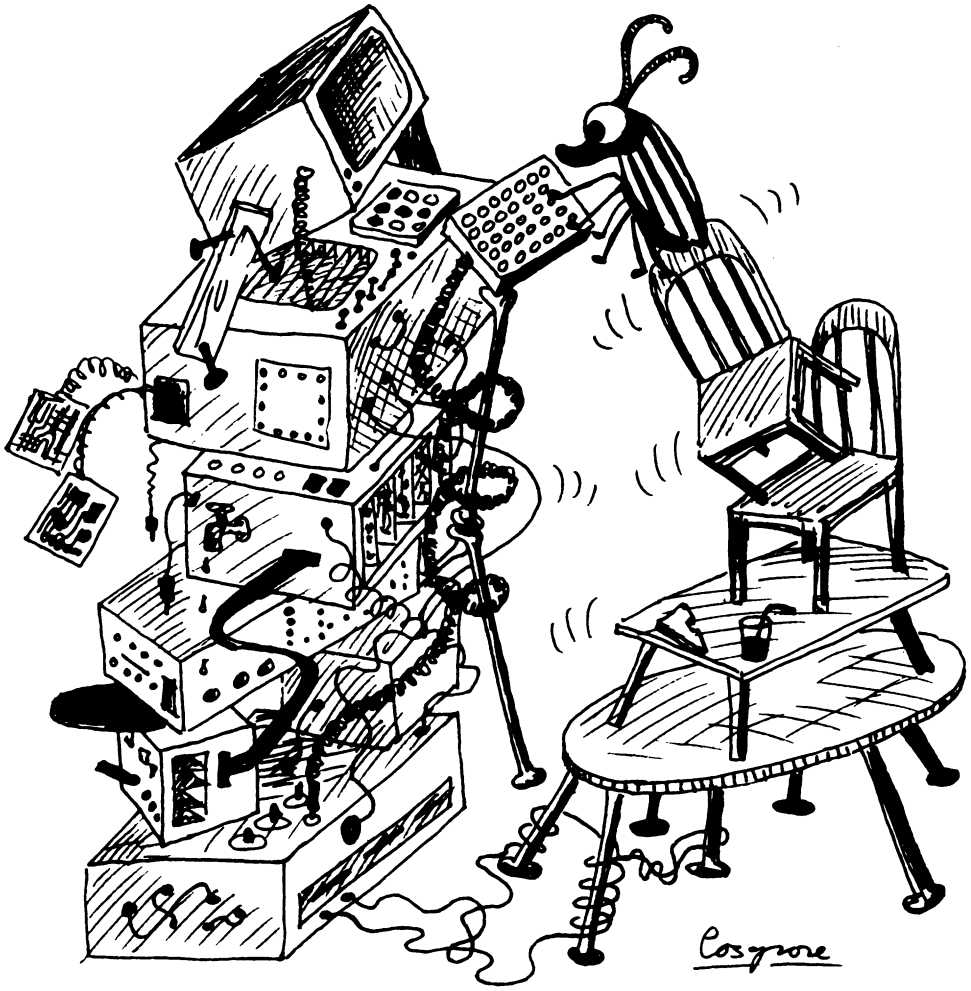
To the outside user, our Nim or Othello player will appear to play pretty sensibly. It will avoid obvious traps, go for winning moves and, in general, will not make aimless plays. That's what a human player would do. So, does the program exhibit intelligence?

Well, it depends on what you mean by intelligence. If you mean the ability to model human behaviour in some way (however restricted the field), then the program does do that. If you mean the ability to solve problems which are completely new to the machine, then of course, it can't. But can anybody? Don't we all need some set of rules to work from—some prior knowledge? So maybe the real problem is to devise ways of giving the machine sets of rules (a model of the relevant aspect of the world, if you like) in as convenient a way as possible.

The family tree example is, if you think about it, a simple case of rule giving. After all, a statement like: "Martha and Tom are the parents of Jim" is not, logically, very different from a doctor making a diagnosis saying "A high temperature and spots indicate measles". All we've done is replace "Martha" with "a high temperature", "Tom" with "spots" and "Jim" with "measles". It's possible to design a system which allows a consultant surgeon, for instance, to tell it about symptoms, diagnoses and so on, and which stores all this information in tree-like structures. It can then search the tree to make its own diagnoses when sets of symptoms are given to it. It may even "discover" simplified sets of rules leading to a particular conclusion.

So trees are pretty interesting structures, although handling them can get a little hairy at times. Usually, though, splitting the program down into manageable lumps (as we did in the Nim program), and careful dry running, will allow you to create interesting and useful routines.

Structured Programming



As well as structuring your data, you should also structure the *program*. Essentially this means writing it as a series of linked subroutines—which can be debugged and tested separately, and then strung together, safe in the knowledge that they work. You can name the subroutines in REM statements if you want, to make the listings easier to follow.

More elusive than structure is *style*. Each programmer has his own; and *my* favourite trick may not be to *your* taste. If you compare FRENCH COUNTDOWN and ZEDTEXT you'll certainly notice the *difference* in style—but you may not prefer the same style as somebody else. Style is partly clarity and efficiency, and partly good organization; but there's something else, which is hard to pin down.

Rather than lecture to you about structure and style, I've written out three case-histories of the design of longish 16K programs (in BASIC). One is a text-editor, one a statistical simulation of supermarket checkouts, and one is an educational program to help teach French (kindly provided by Eric Deeson of the Educational ZX User's Group, EZUG). If you don't want to wade through the description, just copy out all the lines in order, as they appear. But I'd rather you took them routine by routine, which is one reason why no complete listing is given on its own.

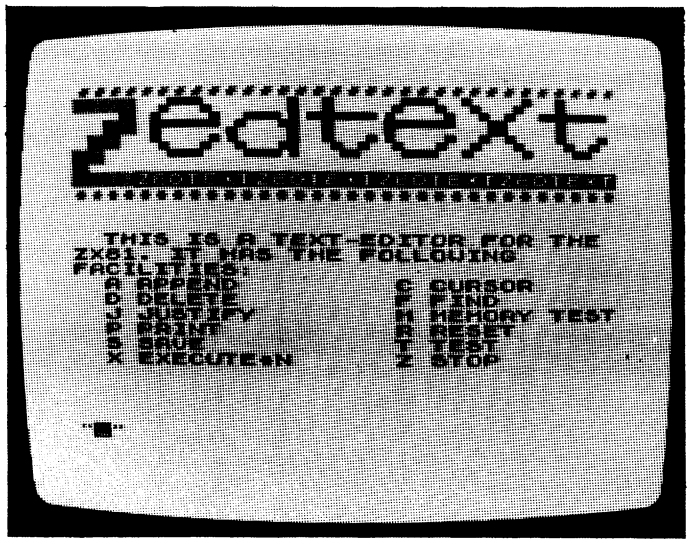
A versatile, albeit slow, text-editor, which gives insight into program structure, word-processors, and the way your BASIC interpreter works.

8 Zedtext

You hear an awful lot these days about *word-processors* or *text-editors*: programs that turn a microcomputer into an intelligent typewriter able to edit and manipulate a written text. The ***1000*** 's facilities for on-screen editing of programs (line cursor [>], character cursor [L], RUBOUT, EDIT, and insertion of new characters from the keyboard) amount to a rudimentary text-editor. Some commercial systems are immensely complicated, and can paginate, justify, and index text automatically.

The ***1000*** won't act as a practical text-editor, at least not until someone produces a gadget to link it up to a fancy printer. But, by seeing how a text-editor for the ***1000*** can be designed, you can begin to appreciate just what is involved in a commercial system. In addition, it's an excellent example of structured programming.

This case-history is the tale of ZEDTEXT and how it came about.



ZEDTEXT title page.

CHOOSING A DATA STRUCTURE

A text-editor must accept textual input, and *process* it in various ways. Before starting on the program to do this, we must first work out the most suitable structure for the data—the text itself. But before structuring the data, we have to consider what we intend to do with it. Vicious circle? No, because we can think pretty loosely at first, and write a few experimental programs to test our hunches.

One important feature is that we should be ooble to delete mistakes like that one, to produce a clean text: that is, we should be able to correct mistakes. Another is that we should be able to append text (or insert it in the middle like this) to existing text, just as this sentence is appended to the previous one. A third is that we should be able to *search* a text to find the first (second, third, . . .) occurrence of a given sequence of symbols—just as the italics find the first occurrence of "search" on this page (and the quotes find the second!). We'd also like to be able to repeat such operations a specified number of times (e.g. "Find the first occurrence of 'FRED' and replace it by 'JIM'; repeat 73 times", which replaces every occurrence of FRED by JIM for the first 73 occasions FRED appears). Numerous other features would be desirable—the ability to remember a segment of text and copy it into a chosen place, the ability to save the results of all this on tape, and so on—but the above will suffice for deciding on a data structure.

So let's think about the possibilities.

(*a*) Think of the text as a one-dimensional array T$ of single characters:

T	H	I	S	□	I	S	□	A	□	T	E	X	T ...
1	2	3	4	5	6	7	8	9	10	11	12	13	14 ...

(To save space I've drawn it horizontally.) The main problem with this is that to insert material in T$ we need to shift lots of characters up several spaces, using code like:

```
1000 FOR K = TOP TO BOTTOM STEP - 1
1010 LET T$ (K) = T$ (K - 92)
1020 NEXT K
```

(say). We start at the top and work downwards, of course, so as not to erase sections of text not yet moved.

This looks as if it would run rather slowly on a text of, say, 3000 characters: otherwise it has the advantages of simplicity.

(*b*) *Linked lists*. Arrange the text in a list, with pointers to the next character.

Number	Character	Pointer
1	T	2
2	H	3
3	I	4
4	S	5
5	□	6
6	I	7
...	... ...	...

Modifications of the kind "insert" or "delete" are now very easy. For example, to change the text to

THIS□IS□NOT□A□TEXT . . .

all we do is add to the *end* of the list

15	N	16
16	O	17
17	T	18
18	□	19

and then modify *two* pointers to get the characters in sequence:

8	□	15
18	□	9

Deletions are equally easy: you just change *one* pointer to reroute past existing text. (One minor problem: the deleted text is still "there" in the sense that it's taking up memory space—it's just that you won't *find* it by tracking through the list following pointers, because nothing points *into* it. If memory is short, it's a nuisance wasting it in this way. There's a good way out, though: put up some kind of flag in the deleted entries, and every so often do a "garbage collection run" which goes through the list re-numbering everything and getting rid of the flagged entries. But such a garbage collection will be slow to operate, so the best time to do it is when you get fed up and go off to make some coffee. Moral: don't collect garbage until the memory is close to full.)

(*c*) A variant on (*b*): use linked lists of *words*. Now you'll have a genuine *word-processor*, but it won't be very good at handling symbolic text, e.g. program listings.

The above discussion is rather theoretical. In practice, which of these three structures you use depends heavily on the particular architecture of the machine you are using. So let's see how the ***1000*** copes.

(*c*) turns out to be a bit awkward, because of that *Procrustean assignment* that the *Manual* goes on about: string arrays always have a fixed length of word. This means that you have to dimension the array to fit in the longest word likely to be used, say

```
DIM T$ (500, 20)
```

for 20-character words. But now our sample text above goes in as

1	THIS□□□□□□□□□□□□□□□□□□	2
2	IS□□□□□□□□□□□□□□□□□□□□	3
3	A□□□□□□□□□□□□□□□□□□□□	4
4	TEXT □□□□□□□□□□□□□□□□	5
	.....	

which wastes *acres* of memory . . . Now, by being clever, you could doubtless take care of some of this; but it all takes time and program space, and it smells complicated.

So ditch that one.

Linked lists (*b*) look very attractive, but there's a hidden snag. The pointers take up 6 bytes each, because of the floating-point representation in the ***1000***. (If you've got an old-fashioned ZX80, which only does *integer* BASIC, you're laughing, because two bytes will do the job admirably. Moral: don't despise the ZX80—you always pay a price for extra sophistication.) So each character in the text takes up 7 bytes instead of 1, reducing the size of the text you can handle.

One way out is to *code* the pointer in binary (or hex) using two characters:

xy means 256 * CODE (x) + CODE (y)

Now the pointer takes 2 bytes per entry—3 altogether. That's given us 7/3 of the capacity. The main price we pay is the time spent encoding or decoding the pointers, an operation we will have to do rather often as we track through the list.

In my ignorance, I actually wrote most of a text-editor using this linked list structure

(*b*). It was a bit slow, but it did the job. A lot of commercial editors do in fact use linked lists: for extra speed they're usually written in machine code—a daunting task. But on the ***1000*** there's a better way, because *the **1000** processes strings very efficiently*. For example, it will accept strings of arbitrary length (subject to memory size). There are very few micros that will do this (a limit of 200–250 characters is common) and indeed many mainframe computers do not have this facility in their standard compilers for languages such as BASIC or ALGOL.

As a result, we can use option (*a*) above, with a difference. As it stands, it's a bit clumsy. But the ***1000*** treats a vector of single characters *as if it were a single string*; and the ability to refer to specific characters in the string is built into the BASIC interpreter. That is, if we set

```
LET T$ = "THIS□IS□A□TEXT"
```

then substring *slices* allow us to ask for, say, the 7th character. For instance,

```
T$ (7) = "S"
```

So T$ is effectively the same as the vector

```
(T$ (1), T$ (2), . . ., T$ (N) )
```

where N is its length. Better still, we can also *assign* substrings: if we type in

```
10  LET T$ (3) = "A"
20  LET T$ (4) = "T"
30  PRINT T$
```

we get

```
THAT□IS□A□TEXT
```

as result.

Thus we have the rather alarming option:

(*d*) Store the text as a *single* (long!) string T$.

Why alarming? Well, we don't usually think of the ***1000*** having to deal with strings 3000 characters long.

But in fact it rather likes them.

I did some experiments, with strings of length 600 or so. The first discovery was that PRINT T$ will display a screenful of characters in about a second or less—about *a factor of 100* better than can be done using linked lists (*b*).

The next: inserting a piece of text (with all that shifting of characters) works even faster. For instance

```
LET T$ = T$ (1 TO 73) + "NEW TEXT" + T$ (74 TO 600)
```

takes next to no time; and so does deletion:

```
LET T$ = T$ (1 TO 43) + T$ (92 TO 600)
```

It's even better than that, because you can save a few bytes by omitting the start and finish of the string, like this:

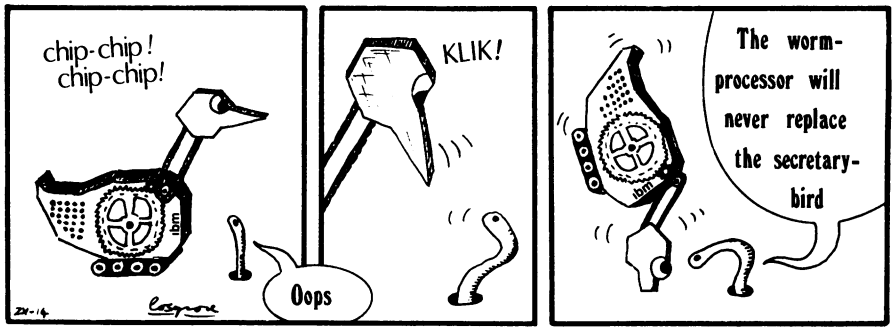
```
LET T$ = T$ (TO 43) + T$ (92 TO)
```

The ***1000***'s BASIC interpreter manages to convert these string-shifting instructions into efficient machine code, hence the speed. For an idea on the kind of code, see pages 153 and 182.

So, after all that, it transpires that there are very good reasons to select a particularly simple

Data Structure: The text will be stored as a *single string* T$ of unspecified length.

As it happens, this has expository advantages too: it's much easier to see how the various subroutines work if this rather transparent data structure is used.



COMMAND FORMAT

The next step is to decide on a format for the text-editing commands that we wish to input. In effect, we are going to write a (small) computer *language*, which the program will *decode* and act upon (*interpret*). Just as the built-in ROM interpreter in the ***1000*** decodes and interprets BASIC language instructions, turning them into machine code, so our program will decode ZEDTEXT commands into BASIC (which the ***1000*** will then turn into machine code—makes you wonder just what *really* goes on inside its little silicon head, doesn't it!).

There is room for a lot of flexibility here. After some thought I settled on the following format for each instruction:

[number] [single letter] [string]

to mean (roughly) perform on the given *string* an operation coded by the *letter*, and do this the indicated *number* of times. For instance

5AFRED

would mean "add to the text five copies of the string FRED". Thus CAT becomes CATFREDFREDFREDFREDFRED, and so on. The advantage of such an instruction format is that it's easy to decode: all you need to do is to scan along it looking for the first non-numerical character, the code letter; then the number part is everything *before* that; the text part everything *after*.

Rather than input a single command at a time, it would be nice to input a whole sequence of commands. To separate the individual commands I needed to choose some little-used character to act as a *delimiter*. For this program I chose "**"** (shifted Q): you can easily change this if you prefer something else. It is set in bold type for clarity. So I was envisaging input commands something like:

5AFRED**"**2FRE**"**7D**"**2ADOG **"**

meaning "*append* 5 copies of FRED; *find* 2nd occurrence of RE; *delete* next 7 characters; *append* 2 copies of DOG". The exact details will come later; the results of course depend on just how the subroutines interpret their commands.

The decoding routine for this "macro-language" must therefore do the following:

1. Scan the string of characters input and locate the first occurrence of the delimiter **"**.
2. Break up the first command into [number] [letter] [string].
3. Use the [letter] code to call a subroutine which implements the relevant command, using the given [string], the required [number] of times.
4. Find the next occurrence of **"**, and repeat the process.
5. Repeat this until the end of the input command is reached.

Here's one way to do it.

```
1000 REM [MACRO DECODE]
1010 LET D = 0
1020 LET I = 1
1030 INPUT M$                                  ┐ read command
1040 LET L = LEN M$                            ┐
1050 IF M$ (L) < > """" THEN LET M$            │
     = M$ + """"                               ├ tidy
1060 LET L = LEN M$                            ┘
1070 LET A = CODE M$ (I)                       ┐
1080 IF A >= 38 AND A <= 63                    │ look for command
     THEN GOTO 1110                            ├ mnemonic letter
1090 LET I = I + 1                             │
1100 GOTO 1070                                 ┘
1110 LET I0 = I
1120 LET K$ = M$ (I)                           ┐ set K$ to the
                                               ┘ mnemonic
1130 IF I0 = D + 1 THEN LET N = 0              ┐
1140 IF I0 < > D + 1 THEN LET N =              │ read number
     VAL M$ (D + 1 TO I0 - 1)                  ┘ preceding K$
1150 LET I = I + 1                             ┐ find next
1160 IF M$ (I) < > """" THEN GOTO 1150         ┘ delimiter
1180 LET D = I                                 ┐ read string between
1190 LET W$ = M$ (I0 + 1 TO D - 1)             ┘ K$ and delimiter
1200 GOSUB F (CODE K$ - 37)                    ┐ carry out command
1210 IF I = L THEN GOTO 1000                   ┘ finish
1220 LET I = I + 1                             ┐ decode next
1230 GOTO 1070                                 ┘ command
```

To see how this works, try it out ("dry-running", see ***Timex/Sinclair 1000***) when M$ is 5AFRED ""2FRE""7D""2ADOG. The first thing it does is stick another delimiter "" on the end, if there wasn't one already. Then it tracks along M$ looking for the first letter, and finds A. It sets K$ = "A". The stuff in front of the A is decoded as a number and set equal to N, so N = 5. Then it tracks along further looking for the delimiter "". Everything between K$ and this is lumped into a string W$, so here W$ = "FRED".

Now armed with the data N = 5, W$ = "FRED", it does a GOSUB to the routine at F (1) corresponding to the mnemonic A. After that it repeats the search from the new delimiter position on, carries out the new command, and continues until it reaches the end of M$. Then it gets ready for the next sequence of instructions.

So, tabulating the results in an obvious way, the upshot is:

Mnemonic letter, K$	Number N	String W$	Operation	
A	5	FRED	GOSUB F (1)	[append 5 copies of FRED]
F	2	RE	GOSUB F (6)	[find 2nd RE]
D	7		GOSUB F (4)	[delete next 7 characters]
A	2	DOG	GOSUB F (1)	[append 2 copies of DOG]

So, assuming we get those subroutines written, if we started with T$ = "" (empty) we would get, in turn:

FREDFREDFREDFREDFRED

FREDFREDFREDFREDFRED

FREDFREDFRED

FREDFDOGDOGFREDFRED

where the underline indicates the "current" position in text (see below for details).

Anyway, it ought to work . . .

CHOICE OF COMMANDS

I decided on the following possibilities for K$:

A append extra text
C move the cursor
D delete
F find a given piece of text
J justify
K list the program
L list the program (yes, again! I kept hitting it by mistake)
M display how much memory is available
P print a "page" of text
R reset all variables
S save on tape, including current text
T set up a test text
X execute preceding commands a given number of times
Z stop

Of these, K, L, M, T and Z were introduced for convenience during the writing—I kept hitting K, for "list", but since I was inputting a string, the machine refused to obey that instruction. T saves a lot of time testing, and is useful for demonstration purposes.

What happened in practice was that I slowly built up the subroutines for these commands, fitting them in at convenient line numbers. By the time the whole program was written and debugged, the numbering had got pretty random and the line numbers were all over the place. At that stage I decided to tidy it all up: but at least I then knew how many lines each subroutine would need. (A line renumbering routine would do a lot of this job, but it's not a universal panacea.) Of course, when you tidy programs you make changes and mistakes, and need to debug again, but it's worth taking the trouble if you intend to use the program often, or want to modify it later. Moral: when the program *works*, you haven't necessarily finished.

Putting the subroutines in alphabetical order (by "mnemonic" K$), and allowing for their lengths, leads to an initialization routine for the array F, like this:

```
10  DIM F (26)
20  LET F (1) = 2000
30  LET F (3) = 2200
40  LET F (4) = 2500
50  LET F (6) = 2600
60  LET F (10) = 2800
70  LET F (11) = 3000
80  LET F (12) = 3000
90  LET F (13) = 3100
100 LET F (16) = 3300
110 LET F (18) = 3500
120 LET F (19) = 3600
130 LET F (20) = 3800
140 LET F (24) = 3900
150 LET F (26) = 4000
```

For example, line 100 means that K$ = P gives a branch to F (CODE K$ − 37) = F (53 − 37) = F (16) = 3300, in the "operating" line 1200 of the MACRO DECODE routine. That is, the PRINT subroutine should begin at line 3300.

Of course, you don't actually need to initialize F *this* way: once these values are stored in memory, they get SAVEd without change; so you can put them in by direct commands, or by writing a little initialization routine which you then erase, or by whatever method you fancy. But remember not to use RUN if you do it that way! Unless memory's getting short, it doesn't matter much.

THE MOVING CURSOR WRITES . . .

If we're going to append or delete text, we'll need to know *where* to do these things. That's what the cursor command "C" is for: it selects a "current" position in the text. For visual display, I decided to use a flashing cursor, changing the relevant character to inverse video and back a few times, and ending on inverse video for clarity. (The program will nevertheless handle inverse video in a text.) So the "C" subroutine, at F (3) = 2200, comes out like this:

Initialize:

```
200 LET T$ = ""        [empty string]
210 LET CI = 0
```

Now the actual routine:

```
2200 REM CURSOR
2210 IF W$ = "0" THEN GOTO 2360
2220 LET CI = CI + N                          ┐
2230 IF CI > LEN T$ THEN LET CI = LEN T$      ┘ crashproof
2240 IF CI <= 0 THEN GOTO 2360
```

```
2250  LET PN = INT ( (CI - 1)/640)
2260  LET CP = CI - 640 * PN
2270  LET LN = INT ( (CP - 1)/32)
2280  LET CN = CP - 32 * LN - 1
2290  LET J = CODE T$ (CI) + 128
2300  IF J >= 256 THEN LET J = J - 256
2310  FOR Y = 1 TO 10
2320  PRINT AT LN, CN; T$ (CI)
2330  PRINT AT LN, CN; CHR$ J
2340  NEXT Y
2350  RETURN
2360  LET CI = 0
2370  RETURN
```

find print position

invert video

flash cursor

In this routine CI denotes the cursor position in the text T$, so that T$ (CI) is the "current" character in the text. The macro command nCjunk determines a number N = n and the routine advances the cursor index CI to CI + N. If this number is not in a suitable range it is trimmed to avoid program crashes (crashproofing). The new value of CI is processed to determine the print position of the character T$ (CI) on the screen, assuming text is displayed in "pages" of 640 characters (20 lines—the other two are kept spare). A flashing cursor is displayed, ending on the inverse video character for clarity.

For instance, suppose the current text is T$ =

THE□MOVING□CURSOR□WRITES

and the cursor is at CI = 7 as shown. Then the command 5C (followed by any string, or none) will advance the cursor to CI = CI + 5 = 7 + 5 = 12 and display a flashing C at this position:

THE□MOVING□CURSOR□WRITES

↑

flash

The number N can be negative or zero, so we can backspace the cursor or leave it where it is. This last option may sound pointless, but it is used in the PRINT routine, for example, because that is actually "print, and display cursor" and we don't want the cursor to *move*.

However, it is important to be able to *reset* the cursor to the start of the text; and line 2210 copes with this, by making use of the fact that W$ is not normally relevant to a cursor instruction, so can be used to carry extra information. Here C0 (or nC0 for any number n) resets the cursor to the front. Another useful addition might be

```
2215  IF W$ = "1" THEN GOTO 2380
2380  LET CI = LEN T$
2390  RETURN
```

This takes CI to the *end* of text (ready to add more). Throughout the program, CI is kept between 0 and LEN T$ to avoid problems with the range of subscripts; and the case CI = 0 usually requires special treatment.

DISPLAY

So far so good; but we haven't got anything on the screen to work with. Let's write that PRINT routine, at F (16) = 3300. The idea is to use the top 20 lines of screen (saving a couple at the bottom for, say, page numbers or helpful messages), so we get "pages" of 640 characters each. With text up to (about) 3000 characters, that gives 5 or so pages.

The command nP obviously ought to mean "print out the nth page". But what about 0P or just P (which is interpreted as 0P)? It would be nice if this meant "print out the *current* page". And that means we need to remember which page we're on. So we'll have CPN as the "current page number". We'd best initialize it:

```
220  LET CPN = 1
```

Now for the subroutine:

```
3300  REM [PRINT]
3310  CLS
3320  IF N = 0 THEN LET N = CPN
3330  LET CPN = N
3340  LET H = 640 * N
3350  IF LEN T$ < H THEN LET H = LEN T$
3360  PRINT T$ (640 * N - 639 TO H)
3370  PRINT AT 21,0; "PAGE□"; CPN
3375  LET N = 0
3380  GOSUB 2200
3390  RETURN
```

Now we've got CPN, it's much better if C0 sets the cursor to the start of the *current* page. The way to do this is to change the CURSOR routine:

```
2360  LET CI = 640 * (CPN - 1)
```

There's another snag to take care of too. Namely, if the variable PN in the CURSOR routine is not CPN − 1, then the cursor gets printed out even though we're on the wrong page! To avoid this, the instruction

```
2255  IF PN < > CPN - 1 THEN RETURN
```

needs to be added.

As you can see, adding new subroutines that *use* old ones can sometimes mean a rethink on the old ones.

APPEND

One more subroutine, and we're in a position to try the program out. It's no use having CURSOR and PRINT but no text!

We want things like 5AFRED to mean "append 5 copies of FRED", that is, insert FREDFREDFREDFREDFRED into T$. But where? The best place seems to be between T$ (CI) and T$ (CI + 1), that is, *immediately after the cursor*. (This is why CI = 0 to LEN T$ is the sensible range: we need to get at both ends.) Where do we want the cursor to go after appending? On the end of the new stuff, probably; ready to append more, or move on to pastures new.

```
2000  REM [APPEND]
2010  IF N = 0 THEN LET N = 1
2020  LET U$ = ""
2030  FOR Q = 1 TO N
2040  LET U$ = U$ + W$
2050  NEXT Q
2060  IF CI = 0 THEN GOTO 2110
2070  IF CI = LEN T$ THEN GOTO 2130
2080  IF CI > 0 AND CI < LEN T$ THEN LET
      T$ = T$ (TO CI) + U$ + T$ (CI + 1 TO)
2090  LET CI = CI + LEN U$
2100  RETURN
2110  LET T$ = U$ + T$
2120  GOTO 2090
2130  LET T$ = T$ + U$
2140  GOTO 2090
```

make N copies of W$

append this

update cursor

SO FAR, SO GOOD?

Now we're ready for a test run. Press RUN; it wants an input. For a start, try

5AFRED□""P

which should print out on the screen

FRED□FRED□FRED□FRED□FRED□
↑

and flash the final space as cursor. It does, so we're not doing too badly; and there's PAGE 1 at the bottom.

Now try C "" 5C "" 2ACAT□ "" P and see if you get

FRED□CAT□CAT□FRED□FRED□FRED□FRED□
↑

with the cursor where the arrow is drawn. Reset cursor (to CI = 0), shift up 5 (CI = 5); append 2 copies of CAT□; print the lot.

Well—if any of our subroutines were wrong, we'd have run into trouble, so it looks pretty good, doesn't it? *But*, there's one feature we haven't checked out; namely, whether everything works on pages 2, 3, 4, and so on. We need a longer text. Type RUN (to clear away the junk accumulated so far); then input

100AABCDEFGHIJK

and go and make the coffee. (Actually it only takes 20 seconds, even in SLOW mode.) When you come back, T$ ought to be 1100 characters long; just ABCDEFGHIJK repeated over and over again.

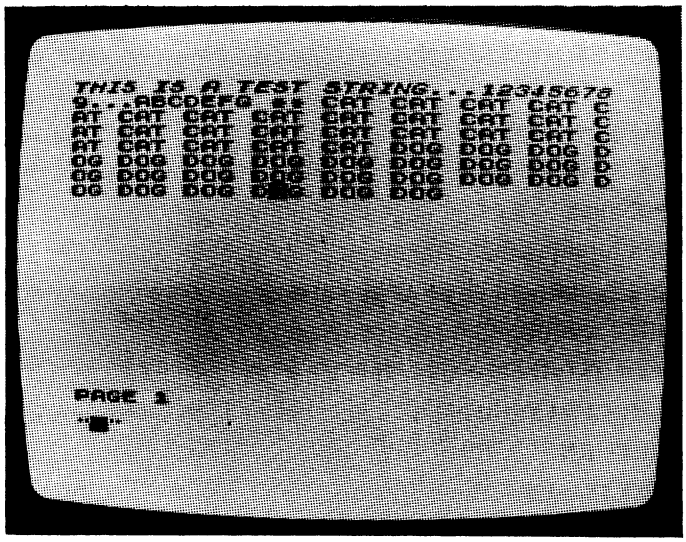
OK, now try

1P

2P

3P

in turn. How are we doing?



*After a test, ending at **: the result of the input command "25ACAT□""25ADOG□""-10C""P".*

DELETE

You should be getting a good idea how all this works by now, so I'm going to leave it to you to test out each new feature as it's added; and I'll try to be more brief.

Command D will delete N characters, starting at and including the current one; and reset CI to the position immediately *before* the deleted passage. Why? Because then it's easy to append text in the space where the deleted bit was. For instance, 8D "" AHARRY will delete 8 spaces, stick HARRY into that slot, and close up the gaps.

```
2500  REM [DELETE]
2510  IF N = 0 THEN LET N = 1
2520  LET L0 = LEN T$
2530  IF CI <= 1 AND N >= L0
      THEN LET T$ = ""                          [empty string]
2540  IF CI <= 1 AND N < L0
      THEN LET T$ = T$ (N + 1 TO)
2550  IF CI > 1 AND CI + N - 1 >= L0
      THEN LET T$ = T$ (TO CI - 1)
2560  IF CI > 1 AND CI + N - 1 < L0
      THEN LET T$ = T$ (TO CI - 1) + T$ (CI + N TO)
2570  LET N = -1
2580  GOSUB 2200
2590  RETURN
```

SOME COME CHEAP

The next few options are trivial to program, but useful—especially during writing and development. I kept hitting K ("LIST") or even L, in the hope of getting a listing—of

course the machine goes to F of something for which there's no subroutine, and all sorts of things can happen. To avoid much hair-tearing, I eventually added in routines K, L, R and Z for LIST, LIST, RUN, STOP. A memory-test M proved useful too.

```
3000  REM [K AND L]
3005  CLS
3010  IF W$ = ""THEN LET N = 1
3015  IF W$ < > "" THEN LET N = VAL W$
3020  LIST N
3025  RETURN

3500  REM [RESET]
3510  RUN
3520  RETURN

4000  REM [STOP]
4010  STOP

3100  REM [MEMORY]
3110  CLS
3120  PRINT AT 5, 0; "SPACE FOR TEXT IS □";
3130  PRINT PEEK 16386 + 256 * PEEK 16387
      − PEEK 16412 − 256 * PEEK 16413;
3140  PRINT "□BYTES"
3150  RETURN
```

This last routine makes use of the system variables ERR-SP, marking the bottom of the GOSUB stack, and STKEND, the top of the calculator stack. However, the machine stack sits in this spare space, so the figure is only an approximation. Still, it will warn you if memory is getting tight.

TEST

This, again, is most useful during writing and debugging; and you can omit it to save space or erase it once the program is working. Use your favourite phrase or saying!

```
3800  REM [TEST]
3810  LET T$ = "TO SEE A WORLD IN A GRAIN OF SAND
      AND A HEAVEN IN A WILD FLOWER,
      HOLD INFINITY IN THE PALM OF YOUR HAND
      AND ETERNITY IN AN HOUR."
3820  RETURN
```

FIND

This command searches the text for the Nth occurrence of a substring W$. So, for instance, 3FBUG asks for the 3rd occurrence of BUG. The cursor should be set at the beginning of the substring BUG once it is found. That suggests we first search for the first

character W$ (1) = B, and then do a subsequent check for UG. To save time, this one is programmed in FAST mode.

```
2600 REM [FIND]
2610 FAST
2620 IF T$ = "" THEN GOTO 2780                    [empty string]
2630 IF CI = 0 THEN GOTO 2636
2634 IF W$ (1) = T$ (CI) THEN LET CI = CI + 1
2636 IF CI = 0 THEN LET CI = 1
2640 IF N <= 0 THEN LET N = 1
2650 LET L1 = LEN W$                              look for 1st character
2660 LET FC = 0
2670 IF CI + L1 - 1 > LEN T$ THEN GOTO 2780
2680 IF T$ (CI) < > W$ (1) THEN GOTO 2700
2690 IF T$ (CI TO CI + L1 - 1) = W$
     THEN GOTO 2720                               look for the rest
2700 LET CI = CI + 1
2710 GOTO 2670
2720 IF FC >= N - 1 THEN GOTO 2750                update counter
2730 LET FC = FC + 1
2735 LET CI = CI + 1
2740 GOTO 2700
2750 LET N = 0
2755 SLOW
2760 GOSUB 2200                                   print cursor
2770 RETURN
2780 SLOW
2785 PRINT AT 21, 0; "NOT FOUND"
2790 RETURN
```

Note that this program looks for the Nth occurrence of W$ *after* (that is, not including) the current cursor position CI.

MULTIPLE EXECUTION

This command—which for simplicity we assume is only used *once* in a given input M$—will mean "do everything before this command N times". That is, 2FCAT""3D ""ADOG""7X"" will mean "find the 2nd occurrence of CAT, delete 3 characters (i.e. delete CAT), append DOG (i.e. replace CAT by DOG); repeat *all this* 7 times".

What we do is set up a counter XC which gets updated by 1 on meeting the command X, until it hits N. Then we reset the counter and continue. Every time X is read, and EC has not reached N, we are sent back to the front of M$ in the MACRO DECODE routine.

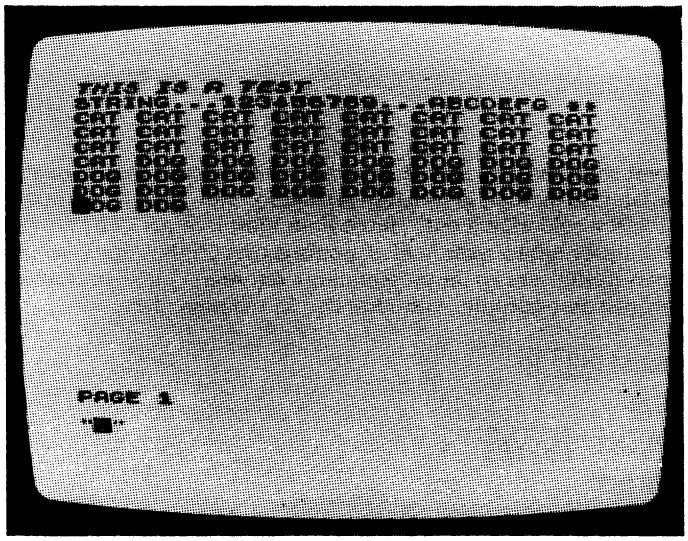
```
3900 REM [MULTIPLE EXECUTION X]
3910 IF XC = N - 1 OR N <= 1          ┐
     THEN GOTO 3960                   ├ finished?
3920 LET XC = XC + 1                  ┐ update counter
3930 LET D = 0                        ┐
3940 LET I = 1                        ├ reset
3950 GOTO 1040                          [yes, GOTO, not RETURN!]
3960 LET XC = 0
3970 RETURN
```

Initialize:

```
230 LET XC = 0
```

Actually this routine leaves some GOSUB addresses piled on the machine stack: if used a lot, it wastes memory. Try to get round this problem by using a GOTO instead of a GOSUB when *calling* line 3900.

JUSTIFY



Justified text: no broken words at right-hand end.

So far the text is printed without any tidying where it goes over a line. The next subroutine "left justifies" the text. ***1000*** lines, at 32 characters each, are too short for right justifying to be worth doing; but you might like to consider what such a routine would look like. (It must distribute the necessary extra spaces between words at random to make up the right length on each line.)

```
2800 REM [JUSTIFY]
2810 LET CI = 1
2820 IF CI + 31 > LEN T$ THEN RETURN          ┐ crashproof
2830 LET V$ = T$ (CI + 31)                    ┘ read line
2835 IF CI + 32 > LEN T$ THEN RETURN
2840 IF T$ (CI +32) = "□" THEN LET T$ =       ┐ tidy next
     T$ (TO CI + 31) + T$ (CI + 33 TO)        ┘ line
2850 LET I2 = 32                              ┐
2860 IF V$ (I2) = "□" THEN GOTO 2920          │
2870 LET I2 = I2 - 1                          │ find last non-space
2880 IF I2 = 0 THEN GOTO 2900                 │ character
2890 GOTO 2860                                ┘
2900 LET CI = CI + 32
2910 GOTO 2820
2920 DIM H$ (32)                              ┐ pad out with spaces
2930 LET H$ = V$ (TO I2)                      ┘
2940 LET T$ = T$ (TO CI - 1) + H$ + T$ (CI + I2 TO)   ┘ stretch T$ to fit
2950 LET CI = CI + 32                         ┐ next line of text
2960 GOTO 2820                                ┘
```

SAVE

The main point with SAVE is that we may want to stop in the middle of an editing job and continue later, so we want to save the variable T$. This, of course, is automatic—but a start with RUN would wipe it out. The obvious thing is to put a GOTO 1000 after the SAVE, so that the program, once loaded, performs the GOTO. Since we're doing that, though, we may as well jazz up the presentation a bit. *And* . . . it's not a bad idea to make the SAVE routine user-friendly, with a few checks and reminders.

```
3600 REM [SAVE]
3610 CLS
3620 PRINT "[SAVE]"
3630 PRINT
3635 PRINT "INPUT PROGRAM NAME. DEFAULT NAME IS
     ""ZEDTEXT"" "
3640 PRINT "START TAPE"
3650 INPUT V$
3655 IF V$ = "" THEN LET V$ = "ZEDTEXT"
3660 SAVE V$
3670 CLS
3680 PRINT AT 3, 10; "[ZEDTEXT]",,,
```

```
3690 PRINT "THIS IS A TEXT-EDITOR FOR THE 1000.
     IT HAS THE FOLLOWING FEATURES

     A  APPEND          C  CURSOR
     D  DELETE          F  FIND
     J  JUSTIFY         M  MEMORY TEST
     P  PRINT           R  RESET
     S  SAVE            T  TEST
     X  EXECUTE * N     Z  STOP"
3700 GOTO 1000
```

You can work out for yourself where to put spaces in the string on line 3690. And you may like to produce more fancy title graphics than those in line 3680—here's one example which you can easily input as a string of graphics symbols.

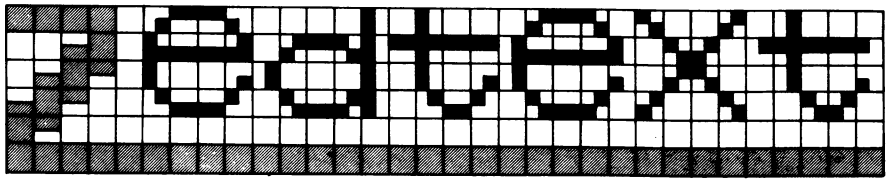


Figure 8.1

DOCUMENTATION

No program is complete without an organized set of user-instructions. If you've been following the way the program was developed, you'll have a pretty good idea how this works; but here's a summary as a reminder. I'll assume you've keyed the program in and SAVEd it.

First LOAD the program using the instruction LOAD "ZEDTEXT" (or whatever name you used). Once loaded, the program will automatically display a title, a short list of options, and then await an input command. Any previously SAVEd text will be in memory.

The text is stored as a single string T$ which, for PRINT purposes, is divided into 640-character segments, called *pages*. A flashing cursor is displayed to indicate the current position in the text.

Instructions to the program take the form [number] [command letter] [string] where the number or string may be omitted in certain cases. Henceforth denote the number by n, the command letter by Y, and the string by w. For example a possible command is 3AFRED, where n = 3, Y = A, w = FRED.

A sequence of commands may be entered at one time by separating the individual commands by the delimiter "" (shifted Q). Thus the most general command takes the form

$$n_1Y_1w_1""n_2Y_2w_2""\ldots""n_rY_rw_r""$$

The final delimiter is optional. To enter a command, type it out and press ENTER.

A sequence of commands is carried out one by one in order. The available commands are as follows.

Number n	Letter Y	String w	Effect
n	A	w	Append n copies of w to the text, immediately after the cursor. If n = Ø or is omitted, the program assumes n = 1.
n	C	—	Advance the cursor n spaces. (n may be Ø, or negative for a backward move.)
—	C	Ø	Reset the cursor to the start of the current text page. (To change the page, use command P.)
n	D	—	Delete the next n symbols, including the cursor position. If n is negative or omitted, the program assumes n = 1.
n	F	w	Find the nth occurrence of string w in the text, starting from the position immediately after the cursor; set the cursor to the first symbol of that occurrence of w.
—	J	—	Left justify the text. (This is effective only on verbal text, with at most single spaces between words. It is best done at the *end* of a session.)
—	K	w	List the program, starting at line VALw. If w is empty, it starts at line 1.
—	L	w	Ditto.
—	M	—	State current spare memory, omitting workspace.
n	P	—	Print page n, and set current page to n.
—	R	—	Reset all variables to initial state (so that the text string T$ becomes empty) and RUN.
—	S	w	SAVE the program under the default name "ZEDTEXT", if w is empty, or w if not. This option includes a reminder to set up the cassette-player.
—	T	—	Set the text to a standard test text.
n	X	—	Carry out all commands in the currently input command sequence n times; then continue to subsequent commands. *This may be used only once in any given input sequence of commands.*
—	Z	—	STOP the program.

[In the above table, the symbol — indicates that n or w may be arbitrary or omitted.]

SYSTEM VARIABLES

The main variables used in the program with consistent meanings are the following.

Numeric

CI	Cursor position in text T$
CN	Used to compute screen position of cursor
CP	Ditto
CPN	Current page number
FC	Counter in FIND subroutine
LN	Used to compute screen position of cursor
N	Current command number
PN	Used to compute screen position of cursor
XC	Counter in MULTIPLE EXECUTION subroutine

String

K$	Current command letter
M$	Current sequence of commands from a single INPUT
T$	Text
U$	n copies of W$ in APPEND subroutine
W$	Current command string

Array

F (26) Lists line numbers of subroutines for command letters: F (1) for routine A, F (3) for C, F (4) for D, and so on by position in alphabet. Entries 2, 5, 7, 8, 9, 14, 15, 17, 21, 22, 23, 25—corresponding to B, E, G, H, I, N, O, Q, U, V, W, Y—are available for user-defined and written subroutines.

POST MORTEM

If you've managed to copy out all of the lines above, and if neither of us has made any mistakes, ZEDTEXT will now be in working order. It's a bit slow, and a bit limited, but it illustrates the way such a program is written.

The speed can be improved by using FAST where possible; but to make any real impact you'd need to go to machine code—a daunting task with something this long, though easier than most things a professional programmer has to do for a living.

The limited range of features can be improved. You've still got mnemonics B, E, G, H, I, N, O, Q, U, V, W, Y to play with: you can add your own subroutines (and initialize F (2) or whatever) as you please. Note that you can do this *without* changing the existing program at all: a big advantage of structuring the beast sensibly.

Even this limited program has features not at first apparent. For example, suppose you want to *replace* every occurrence of JIM by HELEN. You don't have a replace command; but you can use

FJIM "" 3D "" AHELEN "" 999X ""

(where 999 is just chosen to be nice and big). Find JIM, delete it, append HELEN; repeat 999 times (or until you hit the end).

You see why I said ZEDTEXT's macro-language *is* a language? That line of commands is really a little *program* in macro-language, which is interpreted by our BASIC program ZEDTEXT. You can even see what the "program" does, without ever knowing the detailed commands in BASIC. In the same way, the Sinclair ROM interprets BASIC commands into Z80 machine code. So—we haven't just learned about text-editing; we've also learned a little about the way an interpreter works.

Let me expand on that remark, because it's interesting but also potentially misleading. When ZEDTEXT executes a string of Macro Commands, it does it this way:

1. Find the first command.
2. Decode it.
3. Look up the corresponding BASIC routine.
4. Execute that.
5. Find the next Macro Command to be executed (which, in the case of Xn, means either going back to the start of the string or moving on to the next command; and otherwise just means move on).
6. Decode that; repeat until the end.

Thus, although the ***1000*** doesn't speak ZEDTEXT Macro Language as its "native tongue", it's been given a kind of dictionary to translate into BASIC. But the translation is done one command at a time; and on returning to a given command during an Xn loop, it *retranslates it all over again*. It has no way of recognizing "I've been here before".

The BASIC interpreter in ROM works roughly the same way. Each BASIC statement in a program line is "looked up" and turned into Z80 machine code (see later) which the ***1000***'s Central Processing Unit understands. The statement is then executed immediately. The machine then decides which line of program is next to be executed (taking account of FOR/NEXT loops, GOTOs, GOSUBs, and IF/THENs—which makes things hairier in fine detail) and repeats. Again, it doesn't know that it's already decoded a given line before, if it meets it again.

This kind of "translation program" is called an *interpreter*. Like a United Nations interpreter, it reacts mechanically to what is said, sentence-by-sentence; it does not have (or require) any overall grasp of the program structure.

A more sophisticated approach is to use a *compiler*, which turns the whole BASIC program into machine code, but in suitable blocks and not just line by line. A FOR/NEXT loop, for example, would become a corresponding machine code loop; and it would be translated only once; not every time it was encountered during a run. A compiled program usually runs more efficiently than an interpreted one; but the time taken to compile must be offset against any savings. Programs that are going to be used a lot can be compiled once, and thereafter run as machine code, which can be SAVEd as such. A compiler, of course, is harder to write than an interpreter; and the program must be compiled as a whole before it can be run. And even compiled code may not be as efficient as tailor-made machine code. Some BASIC compilers for the ***1000*** are now commercially available; and no doubt there will soon be compilers for other languages like Pascal or Comal.

A simulation of customer-flow in a supermarket, which can be adapted to the allocation of hospital beds and the queues at filling stations.

9 Checkout

This is an example of a computer simulation. A supermarket manager wants to know how many checkouts to keep open at a given time of day, subject to information on the rate of arrival of customers, distribution of time spent dealing with each customer, and so on. The program will attempt to approximate the true sequence of events, keep an eye on queue lengths, waiting times, number of checkouts idle, and so forth, and present its results. This (ideally) would help the manager decide how many checkouts to keep open, and hence how many checkout staff to employ.

The actual program below is just a first step towards a practical simulation; but by the time we've written it we'll be well aware of what extra features it would need, and how to extend it. Meanwhile, we'll keep it relatively simple.

DATA STRUCTURE

What is a good data structure for dealing with N supermarket queues? It's no accident that the word "queue" appears here: the corresponding data structure is designed to do exactly what a real queue does, namely accept input at one end and push things out at the other. What we want is a list of N queues: that is, a *queue array*. Recall that to set up a queue of length 25 (say) we use DIM Q (25), and then special routines to enqueue and dequeue entries. For a queue array we do exactly the same thing, but adding an extra dimension. If CN is the required number of queues (or checkouts) then we'll need DIM Q (CN, 25). The "head" and "tail" pointers H and T, and the lap counter L, also become arrays of size CN.

OPERATION

We'll need a main routine to process the queues, say at time-steps of 1 minute (simulated time, not real time, that is!). At each time step this will have to:

1. Decide how many new customers join the queues.
2. Decide how long each of them will take to pass through the checkout (i.e. how many goods they've bought).
3. Enqueue them according to some reasonable "customer strategy".
4. Decrease by 1 minute the waiting time of the customer at the head of each queue.
5. Dequeue customers whose time has decreased to 0.

In addition, we'll arrange for:

6. The production of a graphic display of the current situation.

After a chosen number of time-steps (say 100) the program should stop, and display an analysis of the way the queues behaved. Some useful quantities would be:

- The average length of a queue.
- The maximum length of a queue.
- The average time a customer spends waiting in a queue.
- The maximum time a customer spends waiting in a queue.
- The average number of checkouts idle.

These quantities will be evaluated by various "system variables", and the main program must keep these up to date; so we'll also need:

7. Update the system variables.
8. Display analysis at end of simulation run.

ENQUEUE AND DEQUEUE

Let's build these subroutines first, because we've already got them worked out (see page 22). Assume we are working on queue number I and wish either to enqueue an extra number V, or dequeue a number and call it V.

Initialize:

```
10  DIM Q (9, 25)
20  DIM H (9)
30  DIM T (9)
40  DIM L (9)
```

We are envisaging a maximum of nine queues here. (To save memory we could first input the number CN of checkouts, then change all 9s above to CN; but there's plenty of room in 16K so why bother?)

```
1000  REM [ENQUEUE]
1010  IF H (I) = T (I) AND L (I) = 1 THEN GOTO 1070
1020  LET Q (I, T (I) ) = V
1030  LET T (I) = T (I) + 1
1040  IF T (I) > 25 THEN LET L (I) = L (I) + 1
1050  IF T (I) > 25 THEN LET T (I) = 1
1060  RETURN
1070  PRINT AT 21, 0; "[QUEUE FULL]"
1080  STOP
```

(If a queue of length 25 gets full, the run stops; but this indicates that queues have grown much too large for comfort, and more checkouts are needed.)

```
1100  REM [DEQUEUE]
1110  IF H (I) = T (I) AND L (I) = 0 THEN RETURN
1120  LET V = Q (I, H (I) )
1130  LET H (I) = H (I) + 1
1140  IF H (I) > 25 THEN LET L (I) = L (I) - 1
1150  IF H (I) > 25 THEN LET H (I) = 1
1160  RETURN
```

As it happens, when we dequeue, V is always 0; but we don't know this at this stage in the writing; anyway, it would be nice to check. We're not worried if a queue gets empty (unlike ENQUEUE where a full queue spells disaster) so line 1110 doesn't ask for a "QUEUE EMPTY" display as line 1070 did.

The initialization of T and H isn't quite what we want, though: the ENQUEUE and DEQUEUE routines assume H and T start at 1, not 0. So we need to add:

```
50  FOR I = 1 TO 9
60  LET H (I) = 1
70  LET T (I) = 1
80  NEXT I
```

GRAPHICS

The next routine sets up CN "checkouts" with spare lines for the queues to be printed out (by another subroutine which follows).

```
1200  REM CHECKOUT GRAPHICS
1210  CLS
1220  FOR I = 1 TO CN
1230  PRINT AT 2 * I - 1, 3; "▄▄▄ "; CHR$ (156 + I)
1240  NEXT I
1250  RETURN
```

To print the queues:

```
1300  REM QUEUE PRINT
1310  LET X = H (I)
1320  LET Q$ = ""
1330  IF Q (I, X) = 0 THEN GOTO 1350
1340  LET Q$ = Q$ + CHR$ (156 + Q (I, X) )
1350  IF X = T (I) THEN GOTO 1390
1360  LET X = X + 1
1370  IF X > 25 THEN LET X = 1
1380  GOTO 1330
1390  PRINT AT 2 * I, 7; Q$; "□"
1400  RETURN
```

build inverse video copy of queue

print it

Each queue is printed out in inverse video: the numbers in the queue represent the waiting time for that individual (i.e. the time needed for him/her to pass through the checkout).

CUSTOMER HANDLING

Next, the subroutines dealing with customers. First we need to decide how many new customers join the queues during one time-step (one "minute"). For simplicity I'll use a random number of arrivals, generated in the obvious way, to produce an *average* arrival rate AR. I'll input AR itself later.

```
1500 REM [ARRIVALS]
1510 LET NA = INT (2 * AR * RND)
1520 RETURN
```

Well, that's almost too trivial to have as a subroutine at all; but I'm going to leave it like that in case someone wants a more complicated distribution of arrivals at some later stage in the development of the program. It will be easier to change it if it's a subroutine on its own. NA, of course, is the number of customers arriving.

What do the customers do when they join a queue? They sit and wait. The question is, for how long? Each customer will have bought a certain number of goods, which will take a certain time to process. For ease of display and computation, I'm going to assume that each customer is assigned a waiting (i.e. processing) time that is a whole number of minutes, and at least 1.

For illustrative purposes, suppose we want to produce a distribution of waiting times, so that of every 30 customers, on average:

```
 4 customers take 1 minute  to process
 6    "      "  2 minutes "   "
10    "      "  3    "    "   "
 8    "      "  4    "    "   "
 2    "      "  5    "    "   "
```

Looks like a messy thing to program, but it's not. Take an easier case: suppose that out of every 6 customers, on average, we want

```
1 customer  to take 1 minute
1    "          "  2 minutes
  .....
1    "          "  6 minutes
```

Obviously we "roll a die" to decide the length of time for each customer. That is, we'd use INT (1 + 6 * RND)—see ***Timex/Sinclair 1000***, p. 34. For the more complicated distribution above, we need to concoct a 30-sided die, such that 4 sides are labelled "1", 6 sides "2", 10 sides "3"; 8 sides "4", and 2 sides "5".

Here is such a die:

```
LET D$ = "11112222223333333333444444455"
```

If we can pick a digit out of D$ at random, we can "roll" the die too; and that's not hard to arrange:

```
LET NT = VAL D$ (INT (1 + 30 * RND) )
```

OK, here's the scenario. When a new customer arrives, (s)he is going to select which checkout to go to by some *strategy*. The waiting time is assigned using the "die" D$. For the moment I'm only going to illustrate the simplest strategy, "choose a queue at random"; but you might like to think about modifying the program to allow two others:

1. Choose the shortest queue (or at random from the shortest ones if there is more than one)—I'll add this in later.
2. Choose the queue with shortest total waiting time (which a customer could estimate fairly well by looking at how much stuff is in people's baskets, so it's reasonably realistic).

That gives us two new subroutines:

```
1600 REM [STRATEGY]
1610 LET I = 1 + INT (CN * RND)
1620 RETURN
```

(which decides which checkout a given customer will go to, and calls it I);

```
1700  REM [NEXT WAITING TIME]
1710  LET NT = VAL D$ (INT (1 + 30 * RND) )
1720  RETURN
```

We've also got to initialize D$:

```
90  LET D$ = "11112222223333333334444444455"
```

MAIN PROGRAM

Now comes the bit that organizes all the rest: it works out what happens during the Nth time-step. First we initialize a variable to count the steps,

```
100  LET STAGE = 0
```

Next we arrange for the operator to input a couple of variables that need to be decided before the subroutines can work:

```
300  PRINT "SUPERMARKET CHECKOUT SIMULATION",,,
310  PRINT "INPUT NUMBER OF CHECKOUTS < = 9"
320  INPUT CN
330  PRINT "INPUT CUSTOMER ARRIVAL RATE"
340  PRINT "(AVERAGE PER MINUTE)"
350  INPUT AR
400  GOSUB 1200
```

Now we're all set for the action:

```
500  REM [ONE TIME STEP]
510  LET STAGE = STAGE + 1
520  PRINT AT 21, 20; "STAGE = □"; STAGE
530  GOSUB 1500                          ┐
535  IF NA = 0 THEN GOTO 590             ┘ number arriving?
540  FOR X = 1 TO NA
545  GOSUB 1600                          ┐ choose queue
550  GOSUB 1700                          ┘ find waiting time
560  LET V = NT
570  GOSUB 1000                          ┐ enqueue
580  NEXT X
590  FOR I = 1 TO CN                     ┐
600  IF H (I) = T (I) AND L (I) = 0 THEN GOTO 640
610  IF Q (I, H (I) ) = 0 THEN GOSUB 1100   decrement front
620  IF Q ( I, H (I) ) = 0 THEN GOTO 640    of queue, and
630  LET Q (I, H (I) ) = Q (I, H (I) ) - 1  dequeue if zero
640  NEXT I                              ┘
```

```
650  FOR I = 1 TO CN
660  GOSUB 1300
670  NEXT I
680  GOTO 500
```

print queues

Not bad. But it won't ever stop! If we want to run a specified number of stages, then we'll need to tell the machine how many, and when to come out of the loop from 680 back to 500. So we'll add:

```
360  PRINT "INPUT NUMBER OF STAGES IN RUN"
370  INPUT NSTAGE
675  IF STAGE = NSTAGE THEN GOTO 2000
```

check if run has ended

At line 2000 we'll put a routine that produces an analysis of the main features of the run.

That raises all sorts of interesting questions, because we'll have to keep track of those features as the run proceeds, and so far we haven't built anything into the program that will do this. So let's take a look at the problems involved.

SYSTEM VARIABLES

To keep track of what's happening, we need to set up various new variables, update them as necessary, and use them to calculate the information required in the final analysis of the run.

First, let's decide what we'd like to know. A few suggestions:

ML = maximum length of a queue.
AL = average length of a queue.
MW = maximum time a customer waits in a queue.
AW = average time a customer waits in a queue.
AE = average number of empty queues (i.e. idle checkouts).

Of these, ML and MW are the easiest to deal with. All we have to do is scan the queue lengths (wait times) at each stage (as each customer joins) and increase ML or MW if this number is larger than the current value. So first we have to initialize:

```
110  LET ML = 0
120  LET MW = 0
```

Then we need two arrays W and N which hold the total waiting time and length of a given queue:

```
130  DIM W (9)
140  DIM N (9)
```

(As before, if we INPUT CN earlier, we can DIM W (CN), etc. to save space; but it's not necessary.)

Now for those averages. You have to think quite carefully what they mean. The way to get an average is to total a lot of numbers and divide out by how many of them there are—but you have to total the right things! The sensible way is to keep a running total, and a running count, and divide one by the other.

Thus to get the average length of the queue, we need a running total length TL, which is updated for each queue once per stage. The running total of queues involved, at CN per stage, is just CN * STAGE. So AL = TL/(CN * STAGE). But where do we update TL?

First we initialize it:

```
150  LET TL = 0
```

Then we decide where to find the queue lengths: the easiest place is during the QUEUE PRINT routine:

```
1395  LET N (I) = LEN Q$
```

We haven't finished with TL yet, but let's put the rest of the calculation into an UPDATE subroutine, to be written below; and start working on AW, the average waiting time. Averaged over what? Average *per customer*. A given customer's waiting time is the total waiting time of the queue he joins, at the moment he joins it, *plus* his own waiting time. So most of the updating will get done during ENQUEUE, where the customer joins the queue. Further, we need to keep a check on the total number of customers that has passed into the system, with a variable TOTCUS. And TW will total the waiting times. Sounds messy? Just keep a clear head!

```
160  LET TW = 0
170  LET TOTCUS = 0
```

Updating is done here:

```
642  FOR I = 1 TO CN
644  LET W (I) = W (I) - 1 + (W (I) = 0)
646  NEXT I

1002  LET TOTCUS = TOTCUS + 1
1004  LET W (I) = W (I) + V
1006  LET TW = TW + W (I) - 1
```

UPDATING

The update routine also takes care of the "average number of checkouts empty" variable, by making a running total TE and dividing by the value of STAGE. Initialize:

```
180  LET TE = 0
```

```
1800  REM UPDATE
1810  LET EC = 0                                  ┐ set empty counter EC
1820  FOR I = 1 TO CN
1830  IF N (I) > ML THEN LET ML = N (I)           ┐ update ML
1840  IF W (I) > MW THEN LET MW = W (I)           ┐ update MW
1850  IF N (I) = 0 THEN LET EC = EC + 1           ┐ update EC
1860  LET TL = TL + N (I)                         ┐ update TL
1870  NEXT I
1880  LET TE = TE + EC                            ┐ update TE
1890  LET AL = TL/(CN * STAGE)                    ┐
1900  IF TOTCUS = 0 THEN GOTO 1920                ├ compute averages
1910  LET AW = TW/TOTCUS                          ┘
1920  LET AE = TE/STAGE
1930  RETURN
```

Before we get too carried away with the amazing success of this subroutine, we'd better take care of one minor point: there's no way to get into it yet. We need to modify the main program:

```
672 GOSUB 1800
```

ANALYSIS

Coming down the home straight now. All we need is to work out a few more system variables, and print it all out.

```
2000 REM [ANALYSIS]
2010 CLS
2020 PRINT "□□□□□□□□□ ANALYSIS",,,
2030 PRINT "NUMBER OF CHECKOUTS = □"; CN
2040 PRINT "CUSTOMER ARRIVAL RATE = □"; AR
2050 PRINT "CUSTOMER STRATEGY: RANDOM"
2060 PRINT "NUMBER OF STAGES = □"; NSTAGE
2070 PRINT
2080 PRINT "MAXIMUM LENGTH = □"; ML
2090 PRINT "AVERAGE LENGTH = □"; AL
2100 PRINT "MAXIMUM WAIT □□□ = □"; MW
2110 IF TOTCUS = 0 THEN GOTO 2150
2120 PRINT "AVERAGE WAIT □□□ = □"; AW
2130 PRINT "AVERAGE EMPTY □□ = □"; AE
2140 STOP
2150 PRINT "AVERAGE WAIT □□□ = 0"
2160 GOTO 2130
```

A SAMPLE RUN

Once you've got the above listings typed in (and no doubt re-debugged them, typing errors being almost unavoidable in this game) the simulation is ready. Type RUN. When asked, INPUT the number of checkouts (say 7), the arrival rate (say 4) and the length of run (say 10, since the program runs fairly slowly). You'll get a graphic display like this, changing at each stage according to the way the customers behave. Then, after the 10th stage, comes the Analysis.

Here's a typical series of runs, with the arrival rate AR = 3 throughout, and NSTAGE = 20; all the values CN = 1, 2, 3, . . . , 9 have been tried in turn. (You can of course get the computer to do all this for you, by suitable modifications—but for an exploratory attempt it's not really worth the bother.) The analysis is summarized in a table:

CN	ML	AL	MW	AW	AE
1	QUEUE FULL AT STAGE 15—RUN STOPPED				
2	23	12.675	62	31.711864	0.2
3	15	5.8333333	46	18.44	0.2
4	10	3.9125	27	12.5	0.25
5	11	3.31	31	11.565217	0.9
6	7	1.7	18	7.4418605	2
7	6	1.81642857	15	6.7884615	1.75
8	5	1.03125	16	5.4324324	3.85
9	4	1.0722222	13	4.4444444	3.1

So, for example, with 7 checkouts the longest queue in the 20 stages run was 6; the average was about 1.8, the longest wait was 15 minutes, the average wait about 6.8 minutes, and on average 1.75 checkouts were idle. The extra decimal places provide a totally spurious suggestion of accuracy, and shouldn't be believed.

These are probably barely acceptable figures: you wouldn't want customers to wait much longer than that! The figures with 6 checkouts are a trifle worse; with 8, better on average; with 9 better still. With 5, they're terrible! So you conclude that you need about 7 checkouts.

Of course, 20 is a rather short run; but armed with this rough estimate, you can try longer runs with 7 or 8 checkouts. I took NSTAGE = 100, AR = 3, and got:

CN	ML	AL	MW	AW	AE
7	11	3.5857143	34	11.701961	0.95
8	12	1.875	38	7.9620253	2.85

Now 7 looks less good; and 8 is acceptable. Conclusion: we need 8 checkouts. (Incidentally, a run of 100 stages takes about 6 minutes in FAST, which disables the graphics; by cutting out display routines you could reduce this a little.)

ALTERNATIVE STRATEGIES

You won't always get exactly these figures, of course, because of the random element in the simulation; but they're fairly typical. However, you may well suspect that the figures are unreasonably distorted by the choice of customer strategy: in practice, shoppers do *not* choose the checkout to join at random!

We can easily modify the existing program to permit different strategies: simply change the subroutine at 1600, or better still, allow a range of strategy subroutines and call the required one when starting the run. Note that it is this easy precisely *because* we have broken the program up into subroutines.

For example, suppose the customer sizes up the queues, sees which are shortest, and joins one of the shortest ones (at random if there are two or more). Then we would need a new strategy subroutine; and for ease of modification we'll put it at line 3000. Here's the guts of it:

```
3000  LET Z = 1
3010  LET MQL = 25
3020  FOR I = 1 TO CN
3030  IF N (I) < MQL THEN LET MQL = N (I)
3040  NEXT I
3050  FOR I = 1 TO CN
3060  IF N (I) < > MQL THEN GOTO 3090
3070  LET B (Z) = I
3080  LET Z = Z + 1
3090  NEXT I
3100  LET R = 1 + INT ( (Z − 1) * RND)
3110  LET I = B (R)
3120  LET N (I) = N (I) + 1
3130  RETURN
```

find length of shortest queues

list shortest queues

select one at random

As always, there's some initialization to take care of, and a bit of tinkering with the original program. First, DIM B. . .

```
190  DIM B (9)
```

Next, we want to call either our old strategy at 1600 or our new one at 3000. That's easy enough:

```
380  PRINT "CHOOSE STRATEGY: 1. RANDOM,
     2. SHORTEST QUEUE."
390  INPUT STRAT
```

Then we change the branch to STRATEGY:

```
545  GOSUB A (STRAT)
```

We here envisage an array A that holds the strategy subroutine line numbers: if we add even more strategies to the repertoire, we can just modify A. But it needs initializing:

```
200  DIM A (2)
210  LET A (1) = 1600
220  LET A (2) = 3000
```

Finally, we need to tidy the ANALYSIS:

```
2050  PRINT "CUSTOMER STRATEGY: □"; ("RANDOM" AND STRAT = 1) +
      ("SHORTEST □□□□□□□□□ QUEUE" AND STRAT = 2)
```

With more than two strategies, this might get clumsy—*you* can work out a better way!

Incidentally, if you object to all those LETs in the initialization, you can RUN, wipe them out, SAVE, and then GOTO 10. I don't really like doing this, because I have a nasty habit of typing RUN at the wrong moment. A fairly safe way is to add

```
9000  SAVE "CHECKOUT"
9010  GOTO 10
```

and SAVE using GOTO 9000. Then the GOTO 10 is automatic on loading. You could

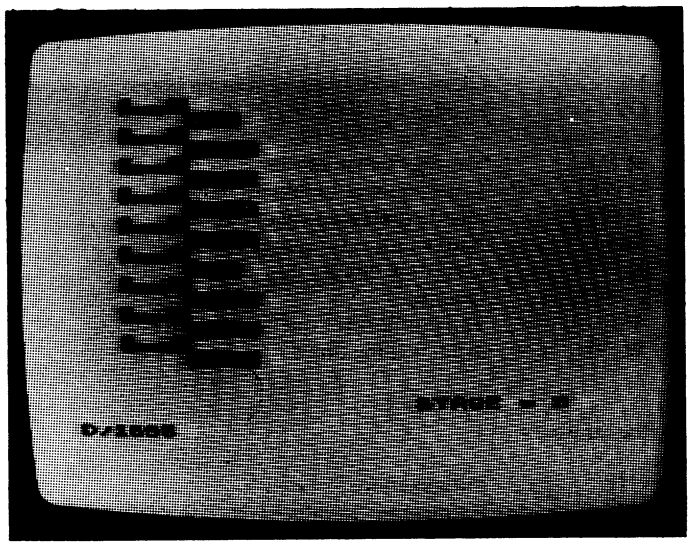
also fit some pretty title graphics in here. It's questionable whether getting rid of all the LETs is really worth it: memory is not at a premium in this program. But automatically running SAVEs is always a nice touch.

Anyway, we now have an additional option in the simulation: the choice of strategy. Let's see what effect it has on the results. RUN as before; but when asked for the strategy input 2. Here's a sample with the same inputs as before: see how the results change!

STRAT = 2, AR = 3.

CN	ML	AL	MW	AW	AE
1	QUEUE FULL AT STAGE 14—RUN STOPPED				
2	10	5.55	35	14.657895	0.05
3	4	2.25	11	10.886364	0.05
4	5	2.6	16	9.9795918	0.65
5	3	1.57	10	7.3695652	0.5
6	3	1.4583333	11	5	1.15
7	2	0.69285714	7	5.7272727	2.95
8	1	0.4375	4	3.6666667	4.5
9	2	0.61666667	6	2.1698113	4

This time we get acceptable figures with only 4 or 5 queues. Note the dramatic jump in empty checkouts (AE) between CN = 5 and CN = 6. Again, the shortness of the run (20 stages, as before) may be causing distortions, so you should now try CN in the range 4-5-6-7 with, say, 100 stages. (I suggest FAST again!) Experiment, and imagine you're the Manager deciding how many staff to hire. Or possibly the Consumer Watchdog, deciding whether the times are acceptable to customers . . .



CHECKOUT display: customers in queues at nine checkouts, with variable waiting times. The queue lengths are almost equal, because customers here have chosen to join the shortest queue.

DOCUMENTATION

The program is pretty much self-explanatory; but here's a summary of its main points.

LOAD "CHECKOUT" and RUN (unless you've done the self-running version). INPUT, when asked, the number of checkouts, average rate of arrival of customers, desired length of run, and chosen customer strategy (1 = random choice of queue; 2 = choose only among the shortest queues). Use FAST to disable graphics and speed calculations, if desired.

The program will perform the simulation, and provide an analysis of the results, listing the above variables, and also:

The maximum queue length.
The average queue length.
The maximum waiting time per customer.
The average waiting time per customer.
The average number of checkouts empty.

System variables

Numeric	AE	Average number of checkouts empty
	AL	Average length of queues
	AR	Customer arrival rate (number per minute)
	AW	Average waiting time
	CN	Number of checkouts
	EC	Counter in UPDATE routine
	ML	Maximum length of queue
	MQL	Minimum length of queue
	MW	Maximum waiting time
	NA	Number of customers arriving
	NSTAGE	Number of stages in simulation run
	NT	Next waiting time
	STAGE	Current stage in run
	STRAT	Customer strategy number
	TE	Running total of empty checkouts
	TL	Running total of queue lengths
	TOTCUS	Running total of customers arrived
	TW	Running total of waiting times
	V	ENQUEUE or DEQUEUE variable
Strings	D$	Dice—distribution of waiting times
	Q$	Used in QUEUE PRINT routine
Arrays	A (2)	Strategy line numbers
	B (9)	List of queues of minimum length
	H (9)	Head pointers for queue routines
	L (9)	Lap counters for queue routines
	N (9)	Length of queues
	Q (9, 25)	Queues
	T (9)	Tail pointers for queue routines
	W (9)	Total waiting times

Subroutine line numbers

10	Initializations
300	Request for variables CN, AR, NSTAGE, STRAT
500	One time step—main program
1000	Enqueue
1100	Dequeue
1200	Checkout graphics

13Ø0	Queue print
15Ø0	Arrivals
16Ø0	"Random" strategy
17Ø0	Next waiting time
18Ø0	Update
2ØØØ	Analysis
3ØØØ	"Shortest queue" strategy
9ØØØ	Autosave

Projects

If you've followed the instructions this far, you ought to be able to tinker with the program without much trouble; or even insert new subroutines allowing extra options (but be careful—you don't want to spend twenty minutes setting things up for each run!). Here are some suggestions—and some additional simulations you can try.

1. *Change the waiting times*. The "3Ø-sided die" D$ controls the distribution of waiting times. Change it, and see what difference it makes. Better still, let the computer build up D$ given the required distribution.

2. *Remove transients*. I've ignored almost completely the problem that we start with an "empty supermarket"—no customers—so the first few stages are not entirely representative. It might be more sensible to run the program for, say, 1Ø stages; reset all variables (except queues and related ones); *then* start the run proper. Try to write a program that allows this.

3. *Multiple runs*. You don't want to sit there all day keying in values 1, 2, 3, . . . for CN. Get the computer to do it; you'll need to store the various analyses in *arrays* now, and modify the final display. Oh, yes—and change the QUEUE FULL jump so that the machine does something useful, rather than just stopping.

4. *Coincident birthdays*. N people at a party compare dates of birth. Ignoring the year, what are the chances that at least two of them have the same birthday?

Simulate it. You'll need a 365-sided die (don't use a D$; use 365 * RND). Don't worry about leap-years unless you're a perfectionist. Generate N birthdays, compare, count coincidences; repeat 5Ø times (say); print the results.

Try N = 5, 1Ø, 15, 2Ø, 25, 3Ø, . . . What size must N be for the chances to be more than even (probability ½) that a coincidence occurs? Unless you know the answer, you'll be surprised . . .

5. *Gas station*. This has a line of PN pumps, each capable of accommodating two cars at a time, but serving only one. Arriving cars join one of two queues (for each side of the pumps), which they cannot leave until served. The queues move whenever a space on that side becomes empty. Given randomly distributed arrival and serve times, find maximum and average queue lengths, waiting times, number of pumps idle, etc.

Generalize to PL lines, each with PN pumps.

6. *Hospital beds*. A hospital ward has BN beds. Patients arrive at random, and stay a random number of days. Beds are allocated as they become spare. If there are none, the patient goes on the waiting list. What is the optimum number of beds to ensure that the waiting list doesn't get too long, and that as few beds as possible remain empty on average?

If you think this is a thinly disguised rewrite of CHECKOUT—you're right! But the sensible numbers are different, and you'll need new graphics.

A case history in educational computing: test your French vocabulary and fly to the Moon at the same time!

10 French Countdown

by Eric Deeson

Eric runs EZUG, the Educational ZX User's Group. Here he writes of his experiences at the sharp end of program design and distribution, where theory must sometimes be tempered with practical considerations. Up till now, provided there's memory to spare, I haven't worried too much about saving space. But long programs take a long time to load from tape; and in educational computing particularly, long waits lead to boredom and the whole process can be subverted before the program ever starts running. So here, straight from the horse's mouth (no offence, Eric!) are some valuable tips on program design and a blow-by-blow account of their use; some special tricks to get more from your ***1000***; and some advice about selling and distributing software on the open market.

INTRODUCTION

Everyone knows that doctors at cocktail parties have a big problem. As soon as they mention their job, they are faced with a list of symptoms to diagnose. Teachers tend not to be invited to cocktail-parties; however, they have been known to drop in on the occasional social gathering. As a computing teacher I have the doctors' problem—if another teacher finds out what I do, the reaction is likely to be "Oh, could you write a program to help my 3C with their quantum mechanics?" Or the details of the Spartan wars. Or the two-times table, or whatever. My weak smile is sometimes followed by total inaction; sometimes an idea grows and reaches the stage of several scraps of paper. Very occasionally, after many hours' work, a polished program results. FRENCH COUNTDOWN came about in that kind of way. The story of how it reached a polished version can now be told. I would like to use it to illustrate how one can produce complex software by way of a methodical approach. This *is* a learning program, one that can be (and is!) used by children at home or at school. But the principles of structured program development are generally applicable. There are also various coding tricks described, at least some of which you may find novel and useful.

Any program, I think, has to meet the following criteria.

- It must do the job for which it was designed.
- It must do so efficiently and with minimum effort on the part of the user.
- Once the program is loaded, it should expect no computing knowledge at all. And the user should never be in doubt what to do.
- The screen display must be pleasing—effectively laid out, uncluttered, easy to follow.
- The user must enjoy working with the program.

The designer needs, therefore, to know not only about programming, but about communication. And he/she must know the subject concerned. In the case of a teaching program, that theoretically means that three people form the development team—a competent programmer, a specialist in the teaching of the subject at the level concerned, and someone with an eye for language and layout. Brilliant as I am, I can't do all that for a French teaching program, and I consulted several teachers about different aspects of the software.

PROGRAM DESIGN

Teachers have to think on their feet a lot, fielding sudden questions with (one hopes) effective as well as immediate answers. I suspect, therefore, that computing teachers are more able than most programmers to write a workable program without much planning. However, that applies only to short chunks of code with straightforward objectives. The temptation to rush at a long complex program without thinking *must* be resisted. I've not been able to resist the temptation sometimes—and I've always got in a heck of a mess in the coding and not been satisfied with the end result. We need a structured approach to the design and coding of programs likely to occupy more than a couple of dozen lines or so. A common type of structured development is called *top-down programming*. We can illustrate it like this:

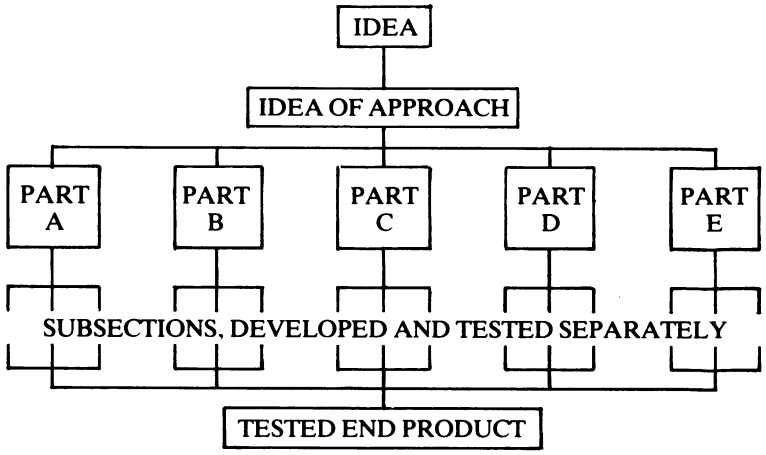


Figure 10.1

Note the word "tested" there. A major advantage of top-down programming is that one can test each individual procedure before linking it with the rest.

In essence, then, we must develop the initial idea into a number of fairly water-tight sections. Each section is called a *module*—it is developed, tested and polished on its own as a subroutine. The final program is a *suite* of such modules, suitably linked together and suitably tested and polished as a whole. Testing may seem irksome, for it is more than just trying the procedure yourself a couple of times to see if it works. The programmer must do his best to make sure it succeeds in *all* conceivable circumstances. If it doesn't, what use is it?

The final program is therefore a set of modules, or subroutines, linked together in some suitable way. We can view it like this:

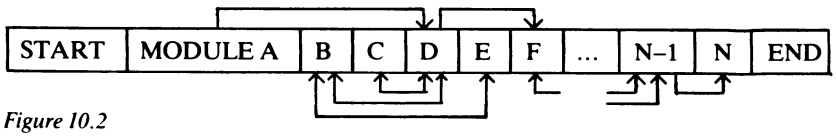


Figure 10.2

Actually I've been using words rather carelessly. It may not really matter here, but there are differences between subroutines, modules and procedures.

A *module* is a self-contained section of a program, one with a logical identity and which can be tested in isolation.

A *subroutine* is a section of code designed to carry out a specific task. There are two kinds. The kind we are used to, called by GOSUB and terminated by RETURN, is a *closed* subroutine. An *open* subroutine, on the other hand, is a set of instructions in a larger one—it still has a specific task but does not use GOSUB/RETURN.

A *procedure* is a sophisticated closed subroutine with its own set of variables. The ***1000*** does not offer procedures in this sense.

From now on, I'll use the word "module" to indicate a conceptual part of the final program. As far as coding is concerned, a given module may be a set of open and/or closed subroutines.

As I do quite a lot of coding, I find it useful to reserve different parts of the computer memory for different types of module. The ***1000*** memory map for my programs is laid out like this:

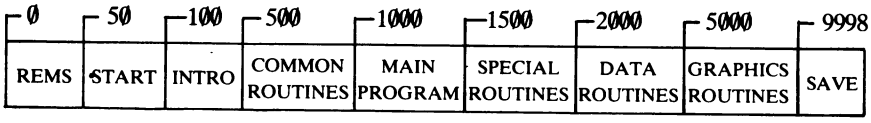


Figure 10.3

You'll see the differences between the sections when we get to look at the development of FRENCH COUNTDOWN.

Here are the stages of the structured modular development of a program. The stages marked * involve particularly careful testing, by the writer, by others, by victims representing the final target population.

1. Definition of program aims and objectives.
2. Development of plan of approach leading to outline flowchart and story-board. (I'll explain these posh terms later.)
3. Analysis of the overall program into logical modules, based (theoretically) on the boxes in the outline flowchart.
4. Developing the main program module (*).
5. Developing, coding, and testing each module and subroutine.
6. Linking the modules together (*).
7. Revising as necessary (*).
8. Polishing the main and subsidiary modules for speed, layout, efficiency (*).
9. Adding the opening and closing routines (*).
10. Polishing and testing the whole thing.
11. Preparing documentation as necessary from the records developed earlier.

Records? Yes—keep the paperwork associated with steps 1–3 above, and during the coding stages maintain vigorously the following lists:

(*a*) Variables and their significance.
(*b*) Addresses of modules and subroutines.
(*c*) Graphics plotting data.

At the end, all this may be condensed into one page or so for filing—perfectly adequate if you keep your REMs in the listing.

Remember that no program is ever perfect—if it is worth keeping it is worth improving every so often. At the very least, you will come to find parts of it impossibly clumsy in the light of your growing expertise—those summary records in the file will make modification fairly straightforward.

DESIGNING FRENCH COUNTDOWN

By now you will have forgotten that I'm supposed to be talking about the development of a specific program—FRENCH COUNTDOWN, a language teaching unit.

How was all the theory of the last section applied in this case? Let's take it step by step.

Step 1: Definition of aims and objectives

The original cocktail-party request was for a program with which individual pupils could test their elementary French vocabulary. That overall plan led to the following aims.

1. To draw at random from a pool of vocabulary items, requesting translation either from English to French or the other way.
2. To keep count of score.
3. To do all this within the context of a simple game.
4. To provide a printout of the correct versions of questions answered wrongly.
5. To set up the pool so that it could be easy for the teacher/parent to change the vocabulary tested.

There was only one specific objective—that when the user had run the program two or three times he/she would know the vocabulary better. (That objective indicates that, in educational jargon, this is a *drill* program rather than one for just testing knowledge. I admit that the original request was for a simple test; trust me to think I know better.)

Step 2: Development of plan of approach

The above aims can be expressed in a very simple flowchart.

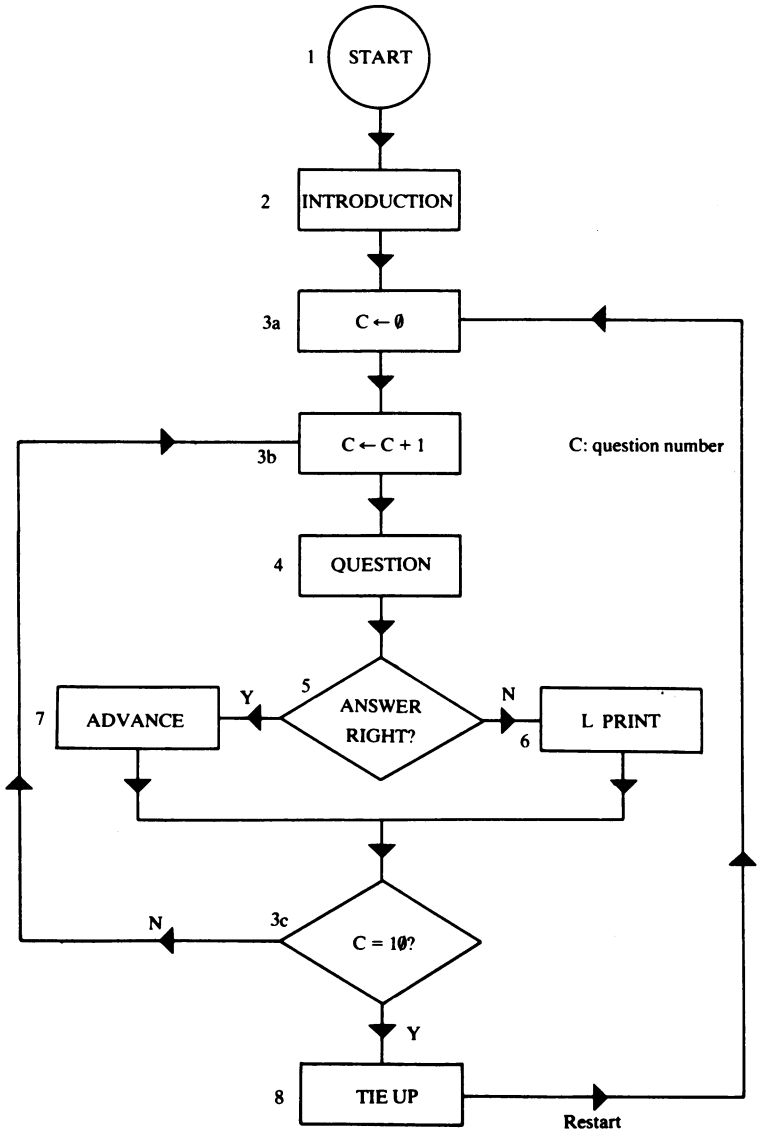


Figure 10.4

The skeleton of a set of modules is appearing out of the gloom already. Some are tiny routines, like the C-loop (setting a given number of questions); others, like getting an individual question on screen, and checking out the answer, are going to be complex and will need much more definition.

What about that ADVANCE box? Therein lies the game aspect of this program. Some kind of "thermometer" display is needed, a graphics routine to show how successful the user is. Many programs of this nature, if they have a graphics game angle at all, stick to getting a train to move further and further along a track. Not me. With an eye on "Space Invaders" I settled on launching a rocket ship. Each correct answer would build the ship up further on the launch-pad, like sketch (a) below. Full marks at the end of the run, and the ship would buzz off around a screen full of stars. Sketch (b) shows that dream. (I didn't succeed in that, I tell you now.)

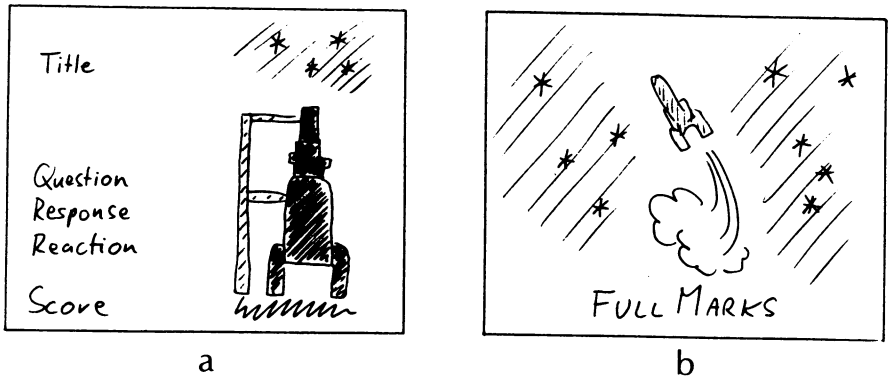


Figure 10.5

Sketches like this form what is called a *story-board* (a term intended to show my keen knowledge of the film industry, another industry in which I have failed to make a fortune.)

Boxes 3, 4, and 5 are going to make the main program modules. 3 is simple—it needs no more extension as it will form a nice little FOR ... NEXT loop. Box 4, though, needs more thought. Here the program must select a question from the pool, check that it hasn't already been used, and present it on screen. To some extent the routines there will depend on the data structure used to hold the pool of questions, but at this stage we can break box 4 down into smaller modules like this.

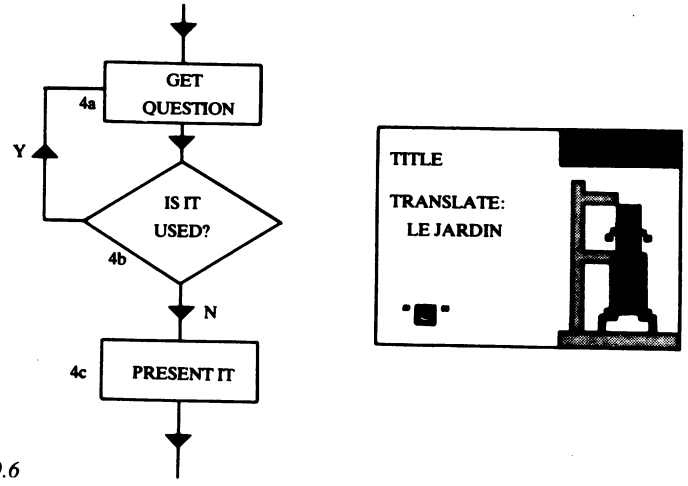


Figure 10.6

Maybe in practice 4a will need further subdivision, in particular to allow random choice of translation direction. Anyway, leave it for now. Same with 5, 6 and 7—they're likely to become complex in practice.

(Actually, at this stage of development, my plans for boxes 6 and 7 were—to say the least—vague.)

CODING

The foregoing took several hours, all without setting a single program line down on paper or in memory. Not only that, but those several hours were spread over several days. That is an accidental part of my program development procedures, for I do have other things to do than sit at a keyboard. But it is now an explicit part of my system. Even when I have a brilliantly exciting program idea I refuse to do any coding until a little while has elapsed in which my subconscious can kick routines around.

But now it's time to code! We're on to steps 4, 5 and 6 at last. First the barest skeleton, based on my standard memory map (sketched earlier):

	Code	*Direct data*	*Variables list*
1	REM [REMS]		
49	REM [START]		
50	SLOW	• LET D8 = 1000	
495	GOTO PROG	• LET PROG = D8	PROG: Start main program (1000)
498	REM [SERVICE ROUTINES]		
999	REM [PROG]		
1005	FOR C = A1 TO B0	• LET A1 = SGN PI	C: Main loop counter
1090	NEXT C	• LET B0 = CODE " ■ "	
1099	REM [FINISH OFF]		
1490	FAST		
1495	STOP		
1498	REM [SPECIAL ROUTINES]		
1999	REM [QUESTIONS]		
4999	REM [GRAPHICS]		
9997	REM [VERSION DATE]		
9998	SAVE "FRENCH COUNTDOWN"		
9999	GOTO START	• LET D1 = CODE "M"	START: Real start (50)
		• LET START = D1	

It's a start! And immediately some points arise:

1. These fragments of code are accompanied with (column two) directly entered data, and (column 3) the growing list of variables.
2. Sprinkle REMs all over the place however good the paper-work. But give them line numbers just below the corresponding start-points—then when REMs are erased the line number flow won't be broken.
3. Also, it's a good idea to put remarks in inverse—boxed in these listings. That makes them show up very clearly on screen or printout.

4. GOTO and GOSUB addresses need not be numbers in Sinclair BASIC— give them meaningful names; four or five characters is enough.
5. To save memory, I code all commonly used numbers—thus D1 (line 9999) is my standard code for 5Ø.
6. The main parts of my programs run in SLOW (line 5Ø), but I code and edit in FAST. So there's a FAST at the end right from the beginning (line 149Ø).
7. Interruptions "occasionally" occur when I'm working, and at the back of my mind is always the dreaded "white out" (which happens very rarely nowadays). So the SAVE routine goes in at once—lines 9998/9. Just before that I keep an updated version REM, so that I don't accidentally restart work on an outdated part-program.

Memory-saving is important. I occasionally do write ***1000*** programs that approach the 16K barrier, but even those that don't are best kept compact to minimize saving/loading time. The main memory-saving trick is to code all common numbers as two-character symbols. We've met three already—here's the complete list. It covers the numbers Ø to 3Ø, so important for loops and PRINT ATs; it also covers numbers commonly used for addresses and delay loop limits. They are entered *directly* in command mode. Using CODE stores numbers in less space than the 6 or more bytes used for floating-point constants.

Number	*Code*	*Entry Code*
Ø	AØ	NOT PI
1	A1	SGN PI
2	A2	CODE " ▗ "
3	A3	CODE " ▄ "
4	A4	CODE " ▖ "
.....		
1Ø	BØ	CODE " ▒ "
11	B1	11
12	B2	CODE "£"
.....		
2Ø	CØ	CODE "="
21	C1	CODE "+"
.....		
3Ø	DØ	CODE "2"
5Ø	D1	CODE "M"
6Ø	D2	CODE "W"
1ØØ	D3	1ØØ
2ØØ	D4	CODE "COS"
5ØØ	D5	5ØØ
1ØØØ	D8	1ØØØ
4E4	D9	4E4

In a program with a lot of numbers, as mine tend to be, this approach saves a huge amount of memory, as much as 10%. It's worth the trouble of direct entering, which I undertake in entirety at this stage. (In fact, I don't—the above coded skeleton, except for the C-loop, plus these and other standard data, are recorded on cassette for use whenever I start a new long program.)

The use of directly entered data means that one must never use RUN (or CLEAR). To avoid the awful hazard of doing that in error, I *always* use GOTO 1 when using *any* program.

Having dealt with the skeleton I must now start building up the main program routine. You see I've allocated only a few lines to this (1000 to 1090). (For FRENCH COUNT-DOWN that proves to be insufficient—showing that I don't really follow my own rules too well.) Without a RENUMBER facility, which I tend not to like with any computer, that means cramped untidy line-numbers. Unless I run entirely out of line number space in a module, I try not to renumber by hand—it's time-consuming and unimportant even if it does make for a neater listing.

I've already dealt with box 3 in the flowchart, so let's get stuck into box 4. The first part of this, 4a, is "get question". As I said before, the procedure depends on how the question data are stored. By now I've decided to have sixty questions in the pool. Elsewhere in this book we describe different data structures—the obvious one to use is a set of directly entered arrays. However, there is an additional criterion to be satisfied, which precludes this—I want the data to be *visible* to the teacher/parent so that they can readily be changed. This means a set of sixty subroutines accessed by

```
GOSUB QUEST + C0 * RND * D2
```

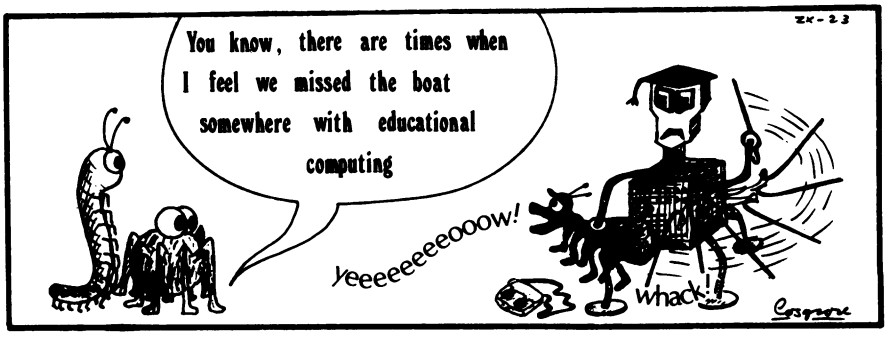
and of the form

```
LET P$ = "LE CHAT"
LET Q$ = "THE CAT"
RETURN
```

May I also draw your attention to the fact that if arrays *were* used to carry the translation items a huge amount of memory would be wasted? This is because the string arrays would have to be dimensioned to enable the longest phrases to be held. (The same problem arose with ZEDTEXT.) If these are "LE PROFESSEUR" and "THE COUNTRY-SIDE" we would need

```
DIM P$ (60, 13)
DIM Q$ (60, 16)
LET P$ (37) = "LE CHAT"
LET Q$ (37) = "THE CAT"
```

and these two last lines alone would take 23 extra bytes. So, my method, as well as giving "teacher transparency", saves some 1½K bytes, and gives another 10% reduction in saving/loading time.



So let's enter the data acquisition lines.

```
1010  CLS
1015  PRINT AT B0, A0; "TRANSLATE:"
1020  GOSUB QUEST +
      INT (RND * D2 + A1) * C0    • QUEST = 2000    QUEST: q section
1025  LET T = INT (RND + .5)                        T: translation
                                                       direction
1030  IF NOT T THEN PRINT TAB A2; P$                 P$: French phrase
1035  IF T THEN PRINT TAB A2; Q$                     Q$: English phrase
1040  PRINT AT C1, A0; "AFTER YOUR ANSWER, PRESS [ENTER] ."
3205  LET P$ = "TRISTE"
3210  LET Q$ = "SAD"
3215  RETURN
```

Run it (with GOTO 1)—it works! Up come, ten times in rapid succession, "TRANSLATE/TRISTE" or "TRANSLATE/SAD", with the ENTER message. Note that line 1020 isn't being tested properly yet—whatever the random value given for the address, the micro arrives at 3205. (A useful trick there.)

That leaves one tricky bit for the module of box 4—only allowing the subroutine to be accessed if it hasn't been used before. What we need is a "flag" system. If each routine starts off with its flag at 0, and this switches to 1 when the routine is accessed, then we can test the flag and try again if it's 1. Here, an array is definitely the structure to use. I'll set up an array U (for "used") with dimension 60; initialize each element to zero; put a line like LET U (N) = 1 in each data subroutine; and then test the flag.

The initialization needs a subroutine of its own. Here it is:

```
100   GOSUB ARRAY    • LET ARRAY = 1600   ARRAY (1600) initialize
                                                  array
1599  REM [ARRAY]
1600  DIM U (D2)                          U: subroutine used flag
1605  FOR N = A1 TO D2                    N: local counter
1610  LET U (N) = A0
1615  NEXT N
1620  RETURN
```

Right—when the program first runs, all those flags will become 0. Try it—GOTO 1 (*not* RUN!), then PRINT U (D1), U (A4) and so on. Then setting the flag:

```
3200  LET U (D2) = A1
```

And then testing for it (a bit of rewriting here):

```
1018  LET N = INT (RND * D2 + A1)         N: local value
1019  IF U (N) THEN GOTO 1018
1020  GOSUB QUEST + C0 * N
```

Get it? Fairly straightforward really. Want to look at the initialization chunk again? Simple—LIST ARRAY. There's another advantage of named routines and subroutines: you can list them by name rather than searching through for them.

This is not going to be so easy to test though, unless you enter a special temporary test

routine. That's a bind and anyway we've got to get all those sixty sets of data in sometime. That I leave to you. You may not want a simple French vocab test anyway. What you've got to do, having worked out what data to test, is to set up 59 more subroutines like that in 3200–3215. FRENCH COUNTDOWN in fact has twenty each of nouns, verbs and adjectives, and lines like:

```
1999  REM [QUEST]
2000  REM [NOUNS]
2020  LET U (A1) = A1
2025  LET P$ = "LE GENDARME"
2030  LET Q$ = "THE POLICEMAN"
2035  RETURN
2040  LET U (A2) = A1
.....
2415  RETURN
2419  REM [VERBS]
2420  LET U (C1) = A1
2425  LET P$ = "FAIRE"
2430  LET Q$ = "TO DO"
2435  RETURN
2440  LET U (C2) = A1
.....
2815  RETURN
2819  REM [ADJECTIVES]
2820  LET U (41) = A1
.....
```

That gives a lot of testing to do—must check the data sometime. So, for the time being change the C-range in line 1005 to TO D2, and then GOTO 1, with an eagle eye on the screen and a talon ready on BREAK. After corrections, use CONT of course. That'll keep you quiet for an hour or so.

But, hold—if we've got twenty each of nouns, verbs and adjectives, can't we get a menu going? Of course we can. Mind you, it's going to mean messing about round 1020 again. Still—it'll be a nice feature.

1. Set up menu, somewhere in the opening section . . .

```
405  GOSUB TITLE          • TITLE = 800        TITLE: 800
410  PRINT AT B0, A2; "I SHALL ASK YOU TEN
     QUESTIONS."                               [Oops, correct 1005]
415  LET A = D1                                A: delay time
420  GOSUB DELAY          • DELAY = 500        DELAY: 500
425  GOSUB TITLE
430  PRINT AT A7, A2; "WHAT TYPE DO YOU WANT THESE",
     "QUESTIONS TO BE?"
435  GOSUB DELAY
```

```
440  PRINT AT B2, A2; "PRESS 1 FOR ... NOUNS."; AT B4, A2; "PRESS 2
     FOR ... VERBS."; AT B6, A2; "PRESS 3 FOR ... ADJECTIVES.";
     AT B8, A2; "4 IS FOR A MIXTURE."
445  IF INKEY$ < "1" OR INKEY$ > "4" THEN GOTO 445
450  LET M = VAL INKEY$                                  M: topic chosen
```

That, an open subroutine by the way, calls two new service subroutines. I use DELAY in most programs. It causes execution to pause for a time set by the parameter A. Unlike the PAUSE instruction, which uses much less memory, it does not give rise to a screen flicker. Like it, though, it can be cut short at any time, by pressing ENTER in this case.

Here's DELAY:

```
499  REM DELAY
500  FOR B = A1 TO A                                     B: delay counter
505  IF CODE INKEY$ = 118 THEN RETURN
510  NEXT B
515  RETURN
```

Yes, you're allowed two returns (or more) in a subroutine.
And here's TITLE, at 800:

```
799  REM TITLE
800  CLS
805  PRINT AT A1, A6; B$; AT A2, A6; A$; AT A3, A6; B$
810  RETURN
```

This employs two directly entered strings—A$, the actual title, and B$, a row of inverse spaces to frame it. So put in A$ = "■FRENCH COUNTDOWN■", with, note, inverse spaces at each end. Then B$ = "18 inverse spaces".

2. After all that convoluted rigmarole (sorry about it) we can LIST PROG and have a go at getting the value of M in use.

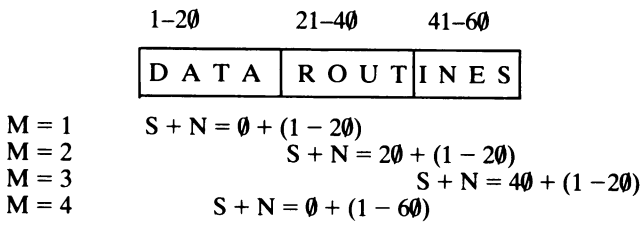
```
1016  IF M = 4 THEN LET N = INT (RND * D2 + A1)
1017  IF M < > 4 THEN LET N = INT (RND * C0 + A1)
1018  IF M < > 4 THEN LET S = ( (M - A1) * C0)          S: data block
1019  IF U (N + S) THEN GOTO 1016
1020  GOSUB QUEST * (N + S) * C0
```

3. Hang on, if you don't see what I'm doing there—there's another addition to make, in the initialization. No, don't groan, it's a little one!

```
1001  LET S = A0
```

Follow that lot through with care. It may seem convoluted, but I think it's fairly neat in the circumstances. (OK, if I'd had the idea of choice of topics before I'd started, I'd have come up with an easier way—but I didn't.)

If M is 1, 2 or 3, line 1018 gets us into the right block (value of S) and line 1017 selects the position in the block. If M is 4, S stays at 0, N is in the full 1–60 range and GOSUB will put us in any part of the data section. This sketch may help.



We now only have to deal with boxes 5–7 in the main program—checking if the answer's correct and acting accordingly. Actually we've been having rather a heavy time, so we'll do box 5 now with just a touch of 6 and 7 and then take a break.

Because of the random choice of translation direction (lines 1025–1035) checking the correctness is not perfectly straightforward. The correct response is Q$ if T is 0 and P$ if T is 1. So:

```
1045  INPUT R$                                          R$: response
1050  IF R$ = " " OR LEN R$ < 3 THEN GOTO 1045          [mug-trap]
1055  PRINT AT C1, A0; E$; AT B2, A0; "YOUR ANSWER:",, TAB A2; R$
1060  GOSUB DELAY
1065  IF (T AND R$ = P$) OR (NOT T AND R$ = Q$) THEN GOSUB RIGHT
1070  IF NOT (T AND R$ = P$) OR (NOT T AND R$ = Q$)
      THEN GOSUB WRONG
```

We've got some more direct data here.

First, E$ is a whole line of spaces, used to wipe out the message printed in line 1040. The easiest way to get 32 spaces is to enter (directly) DIM E$ (32). We also have LET RIGHT = 550 and WRONG = 600. For the moment let's not develop those two subroutines—we can just enter the following and test everything to date:

```
549  REM [RIGHT]
550  PRINT,, "YES"
599  REM [WRONG]
600  PRINT,, "NO"
645  RETURN
```

We'll need to spend quite a bit of time perfecting those routines—but later, after the break I promised.

GRAPHICS

This program is getting quite good already, but it's still only an automatic set-question-and-check-answer machine. The "countdown" in the title concerns the building and launch of a rocket-ship. Remember? So now, to make a change, we'll go fairly fast through the graphics routines from line 5000.

We have ten questions in our main C-loop. So there must be ten stages leading to launch, each stage calling on the predecessors, and the stage reached depending on the score. Part of the RIGHT/WRONG modules will in fact determine score and call the countdown routines. We'll build the graphics up and test it bit by bit and as a whole.

Here's the overall plan.

```
4999 REM [GRAPHICS]
5000 REM [STAGE]
5050 REM [GROUND]
5095 RETURN
5100 REM [SKY]
5105 GOSUB 5050
5145 RETURN
5150 REM [GANTRY]
5155 GOSUB 5100
5195 RETURN
5200 REM [BASE]
5205 GOSUB 5150
5245 RETURN
5250 REM [STAGE 1]
5255 GOSUB 5200
5295 RETURN
5300 REM [STAGE 2]
5305 GOSUB 5250
5345 RETURN
5350 REM [STAGE 3]
5355 GOSUB 5300
5395 RETURN
5400 REM [CLEAR GANTRY]
5405 GOSUB 5350
5445 RETURN
5449 REM [COUNTDOWN]
5450 CLS
5455 GOSUB 5400
5495 RETURN
5499 REM [LAUNCH]
5500 GOSUB 5450
5545 RETURN
```

Can you see the pattern? Each routine calls on the preceding one, which calls on the one before, etc, etc. By the time the score reaches 10 you'll have given your subroutine stack a thorough work-out. This structure should be tested. Insert temporary lines like

```
5055 PRINT "GROUND"
5110 PRINT "SKY"
```

and so on

And then hold your breath and try direct entry: SLOW; GOTO 5500. You should see an awful mess of messages building up—but they should all appear in turn in the right positions.

Now I'll reproduce the designs I came up with, stopping only for really necessary comments. Try to see my short-cuts; run each subroutine when coded by a direct GOTO whatever. Do the testing in SLOW but code in FAST. For clarity gA means "graphics A", etc, □ = space, ■ = inverse space, and boxes denote inverse video.

1. *Ground*

```
5055 FOR K = C0 TO D0                          K: local counter
5060 PRINT AT B9, K; "gA"; AT B8, K; "gA"
5065 NEXT K
```

2. *Sky*

```
5110 FOR K = C0 TO D0
5115 PRINT AT A0, K; "■"; AT A1, K; "■"; AT A2, K; "■"
5120 NEXT K
5125 PRINT AT A0, C1; "[*]"; AT A1, C2; "[*]"; TAB C5; "[*]";
     TAB C7; "[*]"; TAB C4; "[*]"
```

(It's not worth putting those stars in at random or assigning them to a string.)

3. *Gantry*

```
5160 FOR K = B7 TO A5 STEP - A1
5165 PRINT AT K, C1; "[/]"
5170 NEXT K
5175 PRINT AT B1, C2; "[/][/]"; AT A6, C2; "[/][/][/]"
```

4. *Base*

```
5210 PRINT AT B7, C3; "g8□□g5"; AT B6, C3; "g3■g7g7■g4";
     AT B5, C4; "g8■■g5"
```

5. *Stage 1*

```
5260 FOR K = A1 TO A4
5265 PRINT AT B5 - K, C4; "g8□□g5"
5270 NEXT K
5275 PRINT AT B3, C5; "[G][B]"; AT B1, C4; "[/]"
```

6. *Stage 2*

```
5310 FOR K = A1 TO A4
5315 PRINT AT B1 - K, C5; "■■"
5320 NEXT K
5325 PRINT AT A9, C4; "gT■■gY"
```

7. *Stage 3*

```
5360 PRINT AT A6, C5; "gRgE"; AT A5, C5; "g8g5"; AT A4, C5; "g3g4"
```

8. *Clear gantry*

```
5410 PRINT AT A6, C2; "□□□"; AT B1, C2; "□□g8"
5415 PRINT AT A6, C1; " C "; TAB C1; " O "; TAB C1; " U ";
     TAB C1; " N "; TAB C1; " T "; TAB C1; "□"; TAB C1; " D ";
     TAB C1; " O "; TAB C1; " W "; TAB C1; " N "
```

9. *Countdown*

```
5460 PRINT AT B0, A3; " COUNTDOWN "
5465 PRINT AT B2, A7; "■■■■"; AT B3, A7; "■□□■";
     AT B4, A7; "■■■■"
5470 FOR K = A1 TO A9
5475 LET A = C0                                  [gives 1-second ticks]
5480 PRINT AT B3, A9; B0 - K
5485 GOSUB DELAY
5489 NEXT K
5491 IF SC = B0 THEN GOTO 5497                   SC: score
5493 LET C$ = " HOLD —NOT YET READY."            C$: for bottom line print
5495 GOSUB BOTPR                • BOTPR = 750
5497 RETURN
```

I've used a new routine here—BOTPR (bottom line print), one I'm rather proud of. It prints C$ on the *bottom line of the screen*, where the report codes go. Here it is.

```
750 LET X0 = PEEK 16396 + 256 * PEEK 16397       X0: display start
755 FOR X = 761 TO 760 + LEN C$                  X:  bottom line site
760 LET YC = CODE C$ (X - 760)                   YC: character to plot
765 POKE X0 + X, YC
770 NEXT X
775 RETURN
```

What this does, in essence, is to POKE the right character from C$ into the right pigeon-hole in the display file (see Chapter 21). The routine is rather slow, and gives a nice change of pace from the usual printing speed. Do *not* try to mix it with SCROLL, however—I did that once and got the strangest effects—*and* made a whole long program (that I hadn't recently saved) unworkable . . .

Anyway, it's off to launch we go. But, alas, not to realize my dream of having the rocket zoom off into star-set. BASIC's just too cumbersome. Here's how the launch ended up—quite majestic actually, thanks to the vagaries of CLS after SCROLL.

10. *Launch*

```
5505 PRINT AT B3, A8; "0"
5510 FOR K = A4 TO B7
```

```
5515  PRINT AT K, C1; "□"                                    [clear gantry]
5520  NEXT K
5525  PRINT AT B0, A3; "□□□□□□□□□□□□" AT B2, A7; "□□□□";
      TAB A7; "□□□□"; TAB A7; "□□□□"
5530  PRINT AT B6, C5; "**"; TAB C5; "**";
      TAB C0; "□□□□**□□□□"; TAB C0; "□□□□□**□□□□";
      AT C0, C5; "**"; TAB C5; "**"
5535  FOR K = A1 TO D0
5537  SCROLL
5539  PRINT AT C1, C5; "**"
5541  NEXT K
5543  CLS
```

(And here it would have been worth assigning ** to a string variable.)

A lot of care's needed with typing all that from a listing, so check it out properly—it's a nice rocket, and it *does* take off. Remember: use SLOW; GOTO 5500 when you test the whole lot.

POLISHING OFF THE MAIN ROUTINE

After all that work outside the PROG section, you may expect a lot of amendments within it. But no, that's the beauty of the modular approach. There's no effect on the centre of the massive new building in the suburbs.

In fact, we've almost finished the main routine now. All that's left is to deal with the score (incrementing which is of course the function of the RIGHT subroutine).

We need to initialize the score:

```
1003  LET SC = A0                                            SC: score
```

and we need to display it in the C-loop:

```
1075  LET C$ = "YOUR SCORE SO FAR: "+ STR$ SC+
      " OUT OF "+ STR$ C
1080  IF C < 8 THEN GOSUB BOTPR
1085  GOSUB DELAY
```

And that's the end of PROG! Recall that BOTPR is the routine for printing out on the bottom line the contents of C$. Note, by the way, how C$ is structured here, with the use of STR$ to convert a number to a string, and concatenation to join several strings into one. (Even in a PRINT statement, by the way, concatenation is neater than using the semi-colon. Thus PRINT A$ + B$ comes out more smoothly than PRINT A$; B$.)

Just as well deal with 1099 REM [FINISH OFF] now, too. That should make us feel good, though there will still be a few oddments left. Here's how I closed each run of FRENCH COUNTDOWN.

```
1100  GOSUB TITLE
1105  PRINT AT B0, A0; "YOU HAVE HAD YOUR TEN QUESTIONS."
1110  GOSUB DELAY
1115  PRINT,, "YOUR  SCORE  WAS "; SC; "."',,
1120  GOSUB DELAY
```

```
1125  IF SC = B0 THEN PRINT "WE HAVE [LIFT-OFF] —WELL DONE."
1130  IF SC < B0 THEN PRINT "THE COUNTDOWN IS ON [HOLD]."
1135  IF SC < A5 THEN PRINT,, "YOU [MUST] LEARN YOUR WORDS."
1140  LET A = D4
1145  GOSUB DELAY
1150  COPY                                        ["certificate"]
1155  GOSUB TITLE
1160  PRINT AT B0, A0: "LET SOMEONE ELSE HAVE A GO NOW."
1165  GOSUB OPEN          • OPEN = 1500     OPEN [start routine]
1170  GOTO START          • START = D1      START [introduction]
1175  REM [ENDPROG]
```

Nothing remarkable there—except to observe that I didn't concatenate in line 1115. This was because I felt the resulting jerkiness was a bit more suspenseful.

Here's the OPEN routine the above calls on. You may guess from its position that in truth I set it up earlier in the development process than now!

```
1499  REM [OPEN]
1500  LET C$ = "TO TAKE [COMMAND] PRESS [ENTER]."
1505  GOSUB BOTPR
1510  IF INKEY$ = "" THEN GOTO 1510
1515  RETURN
```

Bottom-line print again—yes, I *am* proud of it. And the standard wait-until-any-key-pressed, including ENTER/RETURN, in line 1510.

RIGHT OR WRONG

Now we come to the only remotely hairy bits left—developing the modules in subroutines 550 and 600. These deal, you recall, with correct and incorrect responses respectively, boxes 7 and 6 in the flowchart. For a correct response, we need:

(*a*) to add 1 to the score;
(*b*) a suitable message; and
(*c*) to build the rocket one stage further.

I start off like this:

```
550  LET SC = SC + 1                              [increment score]
```

That's (*a*) done. Trouble with "suitable messages" is that they can be rather gruesome if they don't vary. So I decided to incorporate five suitable messages each for correct and incorrect responses, with random access to them. In this case we have the mini-module RTRES, a set of five subroutines:

```
649  REM [RTRES]
650  PRINT,, "[GOOD]"
655  RETURN
660  PRINT,, "[RIGHT]," + N$ + "."                N$: victim's name
665  RETURN
670  PRINT,, "[CORRECT]"
```

```
675 RETURN
680 PRINT,, " YES ," + N$ + "."
685 RETURN
690 PRINT,, " WELL DONE "
695 RETURN
```

Enter N$ directly for testing purposes at this stage rather than restricting the later development of the introduction.

Now we can go on with RIGHT . . .

```
555 GOSUB RTRES + B0*INT (RND * A5)
560 GOSUB DELAY
565 IF SC < A9 THEN PRINT,, "WAIT FOR STATUS",, "CHECK."
570 LET A = C5
575 GOSUB DELAY
580 IF SC < A9 THEN FAST
585 IF SC > A0 THEN GOSUB STAGE + SC * D1    • STAGE = 5000
590 SLOW
595 RETURN
```

OK? Nothing special there, I think. Oh, yes, the FAST/SLOW bit. In the early versions the rocket was always built up in SLOW— visibly. However, the early guinea-pigs rapidly got bored with that, and the FAST was introduced, except for COUNTDOWN and LAUNCH themselves. The STATUS CHECK message was used to prepare the user for FAST.

Treatment of incorrect responses isn't much different. No increment of score this time of course, but a few extra twiddly bits to make up. Here is the complete module. (If you don't have a printer, disable the LPRINT commands, i.e. ignore them!—though they do no harm.)

```
600 LET U (N + S) = A0                [allows this question to be repeated]
605 LPRINT P$, Q$                     [prints out correct pair of phrases]
610 GOSUB WRRES + B0 * INT (RND * A5)     • WRRES = 700
615 PRINT,, "THE ANSWER IS",, TAB A2
620 IF T THEN PRINT P$
625 IF NOT T THEN PRINT Q$
630 GOSUB DELAY
632 IF SC < A9 THEN PRINT "WAIT FOR STATUS",, "CHECK."
634 LET A = C5
636 GOSUB DELAY
638 LET A = CODE "C"
640 IF SC < A9 THEN FAST
642 IF SC > A0 THEN GOSUB STAGE + SC * D1
644 SLOW
645 RETURN
699 REM WRRES
```

```
700 PRINT,, "[SORRY,]" + N$ + "."
705 RETURN
710 PRINT,, "[THAT IS WRONG.]"
715 RETURN
720 PRINT,, "[NOT SO]"
725 RETURN
730 PRINT,, "[NO, NO], " + N$ + "."
735 RETURN
740 PRINT,, "[TAKE CARE.]"
745 RETURN
```

And all that should be quite straightforward.

THE END

The end is of course the beginning, the only bit we haven't entered. I suggest that that is the right idea—leave the introduction, especially the instructions, until the rest of the program is spotless. I don't think there's anything at all special about the remaining lines. So I'll just rush them off to you, and make one or two asides if need be. First the REMs:

```
0 [ERIC DEESON (C) 1982]
```

And an aside already! You won't want to enter that line, but you can't anyway. Still—here's how to do it.

(*a*) Enter 1 REM [RIC DEESON] etc. (Did you know the Spectrum has a © key?!)
(*b*) POKE 16510,0—this replaces the 1 with 0, so delete line 1.
(*c*) POKE 16513,170—this replaces the REM with [E].
(*d*) Note that the ZX81 can't execute this line, another reason not to use RUN—if you do, you'll get an instant report code!

Anyway . . .

```
10 REM [USE GOTO 50 NOT RUN]
20 REM [T/S 1000 16K]
30 REM [PRINTER UTILIZED]        [If you don't own one, disable the
                                 LPRINT commands]
```

And the introduction itself, with its own little bit of initialization first:

```
60 LET A = C0
100 GOSUB OPEN
105 GOSUB TITLE
110 GOSUB DELAY
115 PRINT AT A5, A2; "DO YOU KNOW YOUR FRENCH WORDS?"
120 GOSUB DELAY
125 PRINT AT A9, A2; "IF YOU DO, PREPARE FOR . . . . . ",,,
    "LIFT-OFF."
130 GOSUB ARRAY                  [gives a few seconds' delay too]
349 REM [START MAIN]
```

```
350  GOSUB TITLE
355  PRINT AT B2, A0; "THIS IS MISSION CONTROL,";
     AT C1, A0; "PLEASE TYPE YOUR NAME AND [ENTER]."
360  INPUT N$
365  PRINT AT C1, A0; E$; AT B2, A0; "WELCOME TO THE
     LAUNCH-PAD,", N$; "."
370  GOSUB DELAY
375  PRINT AT C1, A0; "PLEASE TYPE THE DATE, AND [ENTER]."
380  INPUT D$
385  PRINT AT C1, A0; E$; AT B5, A0; "TODAY IS " + D$ + "."
390  GOSUB DELAY
395  LPRINT A$,,N$,,D$          [headings of print-out—makes them jump!]
```

. . . We've done the next bit

```
455  CLS
460  LPRINT "GROUP□"; M    [print-out test choice]
```

And that, my friend(s), is that. Le programme (yes, that's what *they* call it!) est fini.

By now, of course, you've forgotten my opening pages. But rest assured that I've tried to meet the criteria my cocktail-party friend needs:

- effectiveness;
- efficiency;
- ease of use;
- no computing knowledge required;
- no doubts as to actions required;
- carefully laid out, uncluttered, easy to read screen display;
- enjoyment—well, hopefully.

These were not in fact specifically educational criteria—I've also incorporated:

- full test details and corrections on hard copy;
- carefully determined (and varying) pace with delay-interrupt option;
- full mug-trapping;
- minimum predictability.

Well, it's helped *me* revise my French anyway. Et maintenant . . .

DISTRIBUTION

Having sweated many hours over a hot keyboard to produce an all-time masterpiece of useful programming, one's mind naturally turns to the possibility of making the material available to a wider audience than one's family or captive students.

There are many ways of distributing software in the hope of rich rewards. (They include sticking an ad in the computer press, submitting the material to a software library, or finding a publisher for it.) Whichever one chooses, there is now an extra stage—that of preparing the distribution version. This stage involves several activities—before any of which it is essential to get a full LLISTing. These activities are:

1. Removing REMs—to save memory/loading time and to make it harder for others to follow.
2. Undertaking further techniques for cutting memory/loading time.
3. Polishing the whole thing to impress anyone who sees the coding.
4. Testing again, fully.
5. Renumbering (if you're keen), and testing *again*, fully.

There are, then, two major differences between the master version of a program and the one distributed. The first, and more important, is the minimization of memory requirement. Removal of the REMs so necessary during development, and shortening variable names can, in particular, lead to a reduction of 10–15% in the length of the program and therefore in its loading and saving times.

Another very significant saving can be achieved if the program incorporates arrays which are initialized during execution. It is short-sighted if such arrays have to be saved and loaded. Thus an array A (500) takes over a minute to transfer between micro and tape, even if it's empty. So delete it before saving by using the direct command DIM A (1). As long as the program execution redimensions each array during initialization, this short-cut presents no problem.

The second aspect of preparation for distribution is, I suspect, the more important to those folk who're neurotic about copy-blocking. *There is no way to stop pirates copying your programs*. The method for copying any program is straightforward, but I shan't give it here. (Potential pirates are invited to send me—er—$50 for the secret, if they promise not to tell anyone else.) All the same most people distributing software *do* make some attempt to make it harder to pirate. They may at least make the listing hard to disentangle (well, that's the reason so often given for a cassette full of spaghetti), or they may prevent its being LISTed or LLISTed. If *you* are worried about this, best just "fingerprint" the program by including some dummy lines or directly entered combination code. Then at least you can test a suspected rip-off to see if it bears your fingerprints.

USING FRENCH COUNTDOWN

There are two schools of considered thought about user documentation. (I say "considered" because many people don't seem to give the matter any thought at all.) On the one hand there are the suppliers who feel duty bound to provide sheaves of literature with their programs; few users can follow it, let alone find it relevant. On the other hand, there are the folk who reckon that their programs are self-documenting—and need no accompanying paper. I'm in the latter class. I have over 200 cassettes for the ZX81 alone (and as many again, in total, for the other computers I use). I find almost insurmountable the problem of storing accompanying paper so that it's as accessible as the tapes. And there are few things in life more frustrating than wanting to use a good program and not being able to find the instructions.

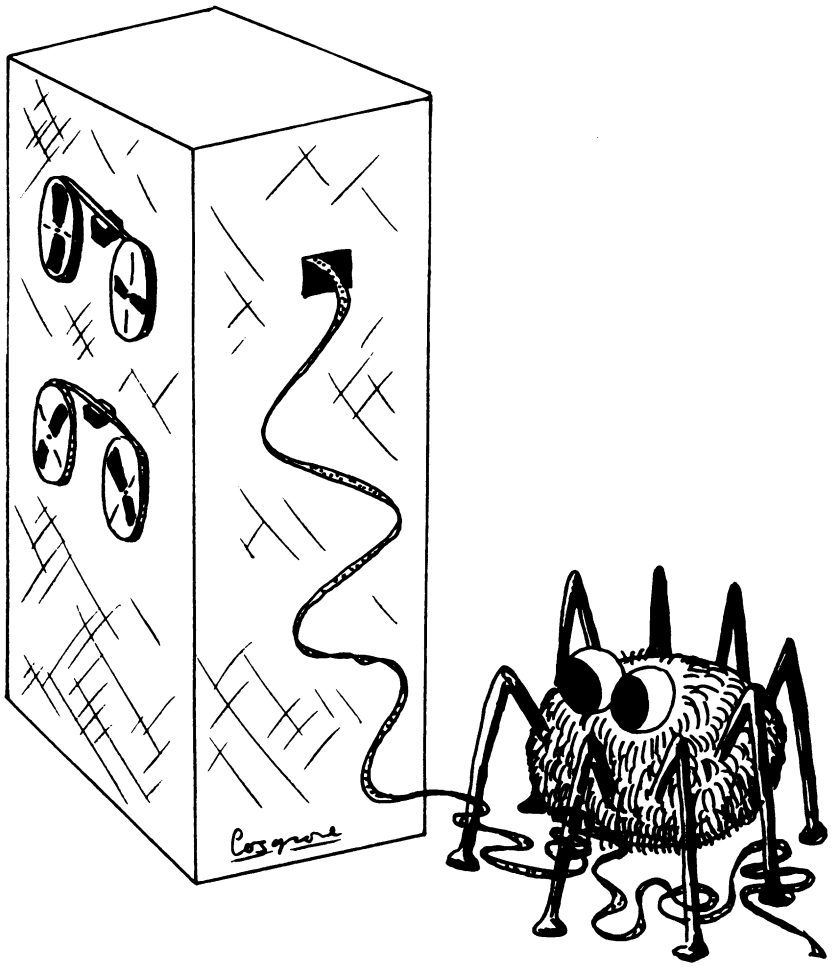
A fully self-documenting program must contain within itself all that the most inexperienced user is likely to need. That user is expected to know only how to get the software into the micro. So a program should be auto-RUN when loaded and should never lead to a report code. Ideally the BREAK key should be disabled—in practice, I mask it stiffly enough so that it's very hard to actuate accidentally. Also the program should contain all necessary instructions. This takes up memory, of course—but that's a great inducement to keep the instructions simple. (Alternative approaches some people use are to have the instructions saved as a separate program on the tape, or to put an audio commentary on the reverse.) I think FRENCH COUNTDOWN is fully self-documenting. Well, almost. So here are all the user instructions it needs . . .

French Countdown (9.8K) recorded on cassette at higher than normal volume for the ***1000*** 16K (printer optional). Valid for any student after some six months of a school French course, this is a game for testing simple English/French and French/English vocabulary. The user is invited to select nouns, verbs, adjectives, or a mixture, and is presented with a test sequence of ten translations. Each correct answer prepares a rocket further for launch; the rocket takes off if all ten questions are answered correctly. The printer outputs details of the user and the test, gives the correct vocabulary pairs for each question incorrectly answered, and reproduces the final score frame. The program is self-starting. Restart after BREAK with GOTO 50, *not* RUN. If delays are found to be excessive, speed them up using ENTER (which is coded [ENTER] throughout the program).

Teachers may wish to change the vocabulary used. It is stored from line 2000.

End of non-self-documentation.

Machine Code



We've been able to tackle some pretty serious problems in BASIC, and I've never actually said anywhere, "This would be a lot easier if we could tackle it in language X", although I may have thought that a couple of times. So why worry with machine code at all? Won't it be much more difficult than BASIC? Is there anything to be gained?

The first question is of course rhetorical. The answer to the second question is that machine code isn't difficult to understand provided you have a clear grasp of the way the machine really handles data, and the form that the data take. The answer to the third is: "It depends." Let me elaborate:

There's a Z80 microprocessor (actually a Z80A, but this makes no difference) at the heart of your ***1000*** which does the real computing donkey work. Unfortunately, it only responds to cryptic, and very simple, instructions written in—you guessed—its *machine code*. Any BASIC statement you want executed has first to be translated into this machine code, and that's done by a program (itself written in machine code) called an *interpreter*, which sits permanently in the Read Only Memory (or ROM) chip of your computer. This translation process takes time, and it's done every time the statement is executed. So if we bypass the interpreter, by writing directly in machine code, we get a dramatic improvement in speed. A program may run something like ten times faster! There are other reasons why speed improvements may be possible, but I'll leave those till later. Of course, whether this increased speed is worth the hassle depends on what you're trying to do. Some moving graphics displays may be hopelessly slow in BASIC. On the other hand, if you're just waiting for an answer to some complicated problem to be printed, you may be quite happy to sit around for 20 seconds rather than 2.

There can even be positive disadvantages to writing in machine code. A program *can* use more memory than its BASIC equivalent. (Again, we'll see why later.)

What I'm saying is that machine code is no cure-all. It's a tool, like any other, to be used in its proper place. If you've ever tried french-polishing a table with a chisel you'll know what I mean.

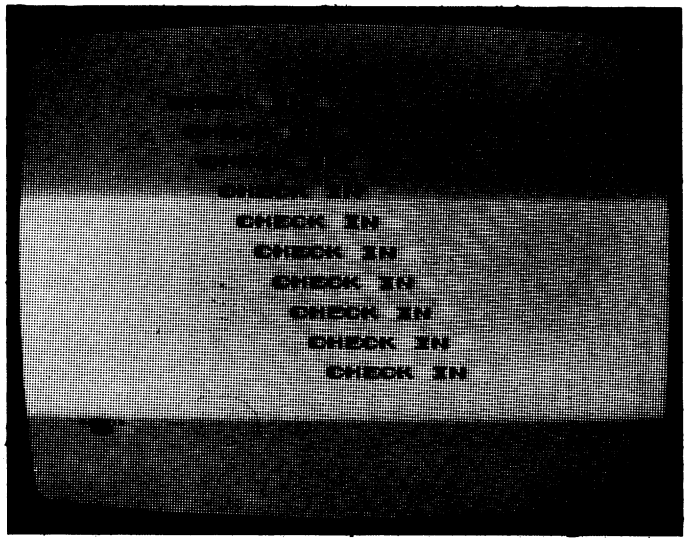
A QUICK CHECK

Since we'll need to wade through a fair amount of stuff before we can write our own machine code and understand what's going on, here's a kind of converse to CHECK-OUT, which I call CHECK IN. Enter it *exactly* as listed. It starts with a weird REM statement that holds the machine code.

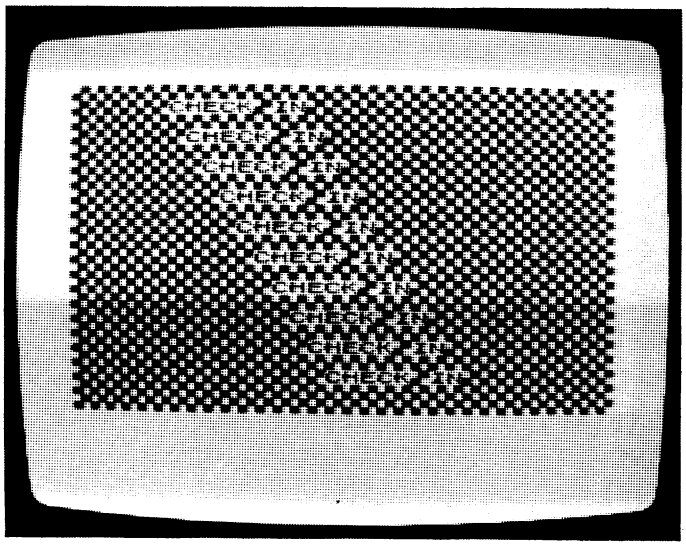
```
1  REM E £ RND 7 X RETURN ■ C ▒ RETURN □ 4 PLOT
   Q ■ / PAUSE TAN
10 POKE 16518, 126
20 PRINT
30 FOR I = 1 TO 10
40 PRINT TAB I + 5; "CHECK □ IN"
50 PRINT
60 NEXT I
70 PRINT AT 21, 31; " ■ "
80 INPUT G$
90 RAND USR 16514
```

The *only* space that is input in the REM is marked; but the machine adds some spaces of its own. The underlined keywords RND, TAN require a function mode, and for RETURN, PLOT, PAUSE you must enter THEN, followed by the keyword, and erase the THEN using Delete. The graphics characters are graphic T and graphic A (characters 6 and 8 by code).

Type RUN. You will get a display, and be asked for a character input. Hit ENTER. Fast, isn't it? You'll never get that sort of speed out of BASIC. The whole procedure probably looks more like magic than anything else, at this stage; but at least you should now be convinced that machine code does have *some* advantages.



CHECK IN: Before . . .



. . . and after.

You can safely change lines 30–60 to produce a different initial display—but do *not* omit line 70. For example, wipe out 30–60 and add

```
30 LIST
```

Experiment. What does the routine do?

When you've finished this book, you'll know that it's not magic at all: in fact you'll be able to write this kind of thing before breakfast. And you should be able to turn back to this page and answer three questions:

1. How does CHECK IN work?
2. Which program later in the book is it pinched from?
3. Why won't it work without line 70?

As I've told you before, the first thing is to understand the data structure. So what structure should *numbers* take in a machine code program?

11 Numbers in Machine Code

I said, a little while ago, that we were going to have to understand how the machine really represents data. Let's start with that.

We normally think about numbers in terms of tens. If I write the number 3814 we all understand that to mean:

$$3 \times 1000 + 8 \times 100 + 1 \times 10 + 4 \times 1$$

and we can see that to get a "place value" from the one on its right we simply multiply by ten. We say the number is in *base* ten.

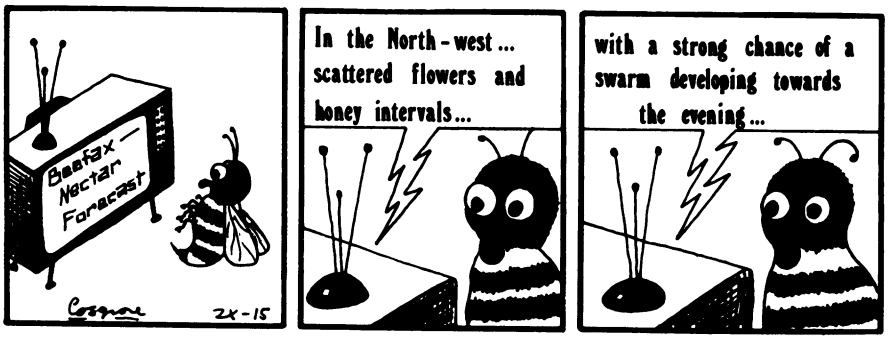
Because we've been doing this for as long as we can remember, it's difficult to realize that there are other, perfectly sensible, ways of doing the same job. Early computer designers certainly didn't; they used base ten representations in their machines and hit some nasty snags. Mostly, they were caused by the fact that electronic amplifiers don't behave the same way for all the signals you want to input to them. For instance, an amplifier that is supposed to output double its input signal may well do so for inputs of 1, 2, 3 and 4 units; but then it starts to "flatten off" so that an input of 5 produces an output of only 9.6, 6 produces 10.8, and maybe you can hardly tell the difference between the outputs for inputs of 8 and 9.

Put a music tape in your cheapo cassette recorder and wind up the volume. Hear the distortion in the loud bits? It's the same effect.

Pioneer computer designers didn't hear any distortion; they just found that the machines couldn't distinguish between different digits at times, and that was hopeless for a computer. So they had to rethink their number representation to suit what the electronic gubbins would do best.

The simplest thing you can do with an electrical signal is to turn it on or off; so you can represent the digits 0 (off) and 1 (on) satisfactorily. Distortion no longer matters. It's clear whether a signal is present or not regardless of how mangled it is. But can we devise a number system which only uses 0s and 1s?

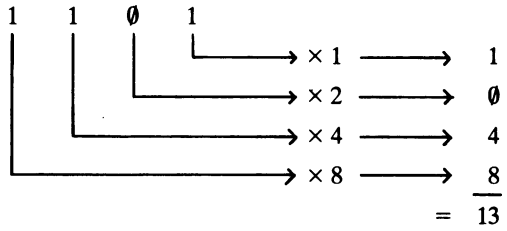
Yes. In a base ten number, the largest possible digit is 9. Add 1 to 9 and you get 10—a *carry* has taken place. We can write any number using any other base we choose, and the



largest possible digit will always be one less than the base. If the base is 2, the largest digit is 1, so a base 2 (or *binary*) number only contains Øs and 1s.

What about the place values? In the base ten case we got those by starting at 1 (on the right) and multiplying by 1Ø every time we moved left one place. For a binary number we still start at 1, but we multiply by 2 every time we move left.

So for instance the binary number 11Ø1 can be converted to base 1Ø like this:



Converting the other way is easy as well; take 25 for example. If we write down the binary place values:

32 16 8 4 2 1

and work from the left, it's clear that we need a 16, which leaves 9, and that's made up of an 8 and a 1, so 25 is:

Ø 1 1 Ø Ø 1

HEXADECIMAL CODE

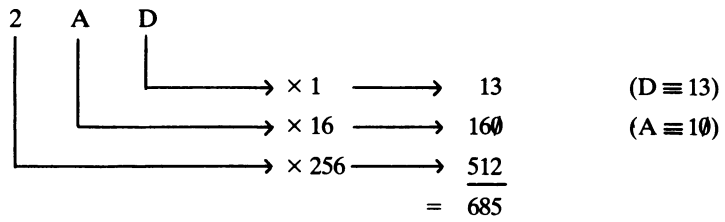
This is fine for relatively small values, but a bit messy for large ones. There are a number of quick conversion techniques, and there are binary-to-decimal and decimal-to-binary conversion program listings in ***Timex/Sinclair 1000***: but I want to examine a procedure which makes use of *hexadecimal* code, because it will stand us in good stead later.

A number in hex (nobody ever says "hexadecimal", except me, just now) is a number in base 16. So the place values are obtained by successive multiplications by 16. The first five are:

65536 4Ø96 256 16 1

"Hang about!" everybody's saying. "Those are nasty numbers, and anyway, in base 16 the largest digit has the value 15. Things are getting complicated."

Bear with me. We handle the problem of digits greater than 9 by assigning the letters A–F to the values 1Ø–15. So the number 2AD in hex converts to decimal like this:



Now for the nice feature of hex. Because 16 is one of the binary place values (the fifth one) it turns out that each hex digit in a number can be replaced by the four binary digits which represent it. (By the way, "binary digit" takes almost as long to say as "hexadecimal" so it's normally abbreviated to "*bit*".) The table opposite shows the conversions:

Decimal	Hex	Binary
Ø	Ø	ØØØØ
1	1	ØØØ1
2	2	ØØ1Ø
3	3	ØØ11
4	4	Ø1ØØ
5	5	Ø1Ø1
6	6	Ø11Ø
7	7	Ø111
8	8	1ØØØ
9	9	1ØØ1
1Ø	A	1Ø1Ø
11	B	1Ø11
12	C	11ØØ
13	D	11Ø1
14	E	111Ø
15	F	1111

A more extensive table is given in Appendix 1.

Now suppose we want to convert 9Ø41 to hex. First we extract two 4Ø96s, then some 256s and so on like this:

```
                 9Ø41
2 × 4Ø96 = 8192 −
                 849
3 ×  256 =  768 −
                  81
5 ×   16 =   8Ø −
                   1
1 ×    1 =    1 −
                   Ø
```

So the hex representation is 2351.

Now we just copy the digit codes from the table:

2	3	5	1
ØØ1Ø	ØØ11	Ø1Ø1	ØØØ1

and that's the binary equivalent of 9Ø41; just run the four blocks together to get ØØ1ØØØ11Ø1Ø1ØØØ1.

The hex-to-binary conversion is so easy that, more often than not, we leave numbers in hex even when, ultimately, we need them in binary. After all, it's easy to make an error in copying long strings of Øs and 1s.

CONVERSION BY COMPUTER

Here's a program to convert from decimal to hex. It successively divides the number by 16, looking at the remainder each time, so it extracts digits in the opposite order to that shown above.

```
  1 DIM HEX$ (4)
 20 LET P = 4
 30 LET HEX$ = "0000"
 40 PRINT "ENTER DECIMAL NO. (MAX: 65535)"
 50 INPUT DN
 60 LET N = INT (DN/16)
 70 LET HEX$ (P) = CHR$ (DN - 16 * N + 28)
 80 LET DN = N
 90 LET P = P - 1
100 IF DN > 0 THEN GOTO 60
110 PRINT "HEX VALUE IS"; HEX$
```

The result is always presented as a 4-digit number, with leading zeros if there are fewer than 4 significant digits. The program won't work if the result should contain more than 4 digits, but that's ideal for our purposes, as we shall see.

Here's the code to convert in the opposite direction (hex to decimal):

```
140 PRINT "ENTER 4 DIGIT HEX NO."
150 INPUT HEX$
160 LET DN = 0
170 FOR P = 1 TO 4
180 LET DN = DN * 16 + (CODE (HEX$ (P) ) - 28)
190 NEXT P
200 PRINT "DECIMAL VALUE IS:"; DN
```

We could tie these routines together with a little menu:

```
 2 PRINT "DEC/HEX CONVERTOR"
 3 PRINT "1) DEC - > HEX"
 4 PRINT "2) HEX - > DEC"
 5 PRINT "3) END"
 6 PRINT "ENTER 1, 2, OR 3"
 7 INPUT SEL
 8 IF SEL = 1 THEN GOSUB 20
 9 IF SEL = 2 THEN GOSUB 140
10 IF SEL = 3 THEN STOP
```

and, of course, we'll need RETURNs at lines 120 and 210.

To deal with negative numbers, the machine uses a clever trick.

12 Positive and Negative

Now that we've seen something about manipulating binary numbers let's return to looking at the way they are handled inside the machine. Usually, a number is held in a fixed number of bits, often 16 or 24 or 32, depending on the machine design. This number of bits is called the *word size* for the machine.

Let's examine what numbers could be held in a 4-bit word:

4-bit pattern	Decimal value
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

It's obvious why bigger word sizes are chosen in practice; a machine which can only represent the numbers 0 to 15 is unlikely to be adequate. But there are two other problems; the notation can't represent fractional values (7.14, for instance) and it can't represent negative numbers.

We'll ignore the fractions problem because most machine code routines only use integers, but the way in which negative numbers are dealt with is more pressing.

The technique is simple: if you've got the binary representation of a positive number and you want to create its negative equivalent you do two things:

1. Change all the Øs to 1s and all the 1s to Øs (this is rather picturesquely called "flipping the bits").
2. Add 1 to the result.

For instance, suppose you want −3.

3 = ØØ11 in a 4-bit word

Flipping the bits gives: 11ØØ
Now add 1: +1
11Ø1

So 11Ø1 represents −3. It's called the *2's complement* of ØØ11.

I'm not going to explain exactly why this works, but you can prove to yourself that it does in any particular case like this:

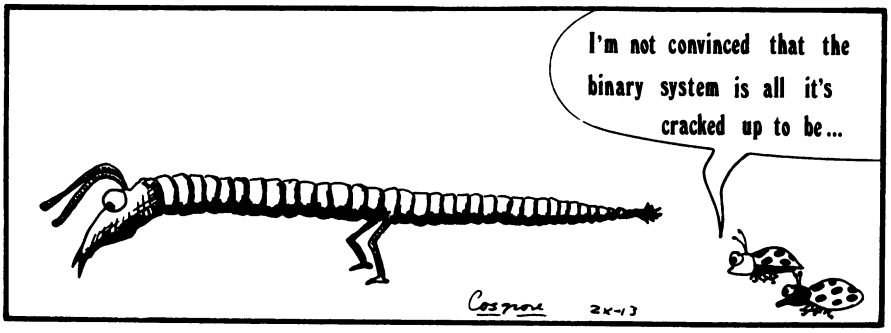
If we add 3 to −3 (or 5 to −5 or anything to minus itself) we should get zero. So:

```
   ØØ11     ( = 3)
+  11Ø1     ( = −3)
=  1ØØØØ
   111      (Don't forget that 1 + 1 = Ø carry 1 in binary!)
```

So we *don't* get ØØØØ at all; but the junior 4 bits *are* zero, and if we're working in a 4-bit word the senior bit will just drop off the end. (For a convenient analogy, think about a car trip-meter with 3 digits; if it reads 999 and you drive an extra mile, it reads ØØØ and a "1" has "dropped off" the left hand end).

In other words we should have seen it like this:

```
   ØØ11
+  11ØØ
   ----
   ØØØØ
   ↓
   1
```



This always works provided that the number of bits is fixed throughout. Don't forget to include leading zeros to make up the number of bits to this standard length, *before* taking the 2's complement.

Let's rewrite the 4-bit table of values, now including negatives:

Decimal	Binary	2's complement	Decimal
0	0000	0000	0
1	0001	1111	−1
2	0010	1110	−2
3	0011	1101	−3
4	0100	1100	−4
5	0101	1011	−5
6	0110	1010	−6
7	0111	1001	−7
8	1000	1000	−8
9	1001	0111	−9
10	1010	0110	−10
11	1011	0101	−11
12	1100	0100	−12
13	1101	0011	−13
14	1110	0010	−14
15	1111	0001	−15

Straight away we see that there's a problem; every bit-pattern occurs twice so that, for instance, 1001 could mean 9 or −7. So we'll have to restrict the range of values still further. I've drawn a dotted line around the region we actually choose to represent. If you look at the senior (leftmost) bit in each of the patterns you'll notice that it's "0" if the number is positive and "1" if the number is negative. This is obviously a very convenient distinction.

So the range of numbers we can get into a 4-bit word is −8 to +7. For 5 bits it would be −16 to +15. For 6 bits it will be −32 to +31 and so on.

A 16 bit word (which is important so far as the Z80 is concerned) holds the range −32768 to +32767. A table of 2's complement notations for 8-bit words is given in Appendix 1.

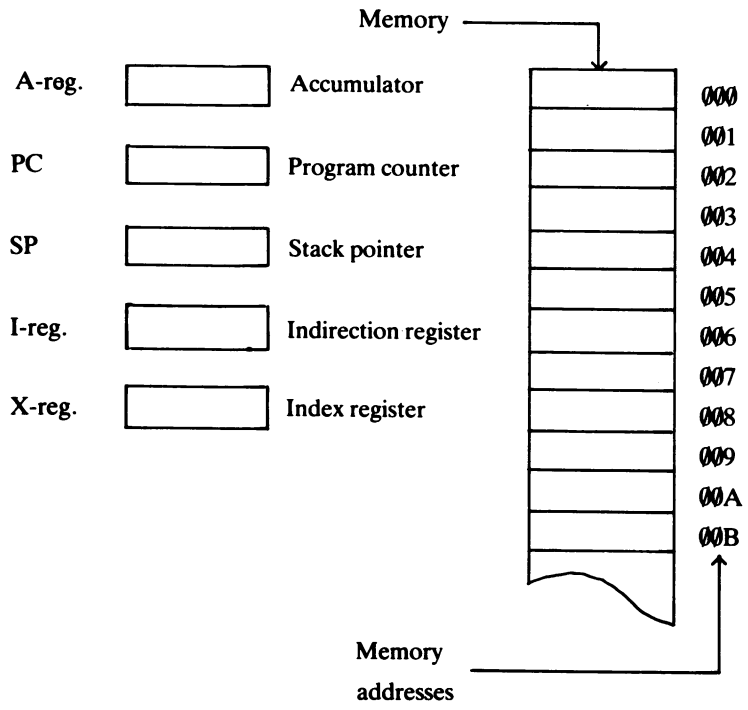
It's easier to start with a simplified, imaginary machine. The Z80 is like this, but more complicated: get the main ideas here!

13 Machine Architecture

That's enough about numbers. Now we'll look at how the machine crunches them. To do this, we need to know about the internal structure of the processor—its *architecture*.

Now, the Z80 processor is the product of some twenty-five years of computer development and is a fairly sophisticated beast. So it's not really a good place for the beginner to start. What I'm going to do, then, is describe a simple processor which might have been built in the late 1940s (except it wasn't), just to introduce the important concepts which are relevant to virtually all current devices, without having to worry about the frills, which we can look at later (in chapters 16 onwards).

We'll suppose that our imaginary machine has a memory of 16-bit words and a number of 16-bit special-purpose registers as shown below:



Let's look at the memory first. In BASIC we could have called each of those memory locations anything we fancied, but the naked machine isn't so friendly. It insists on numbering every location in an absolutely fixed way, starting at zero, as I've shown. These numbers are called the *memory addresses*, and I've numbered them in hex,

although you should always bear in mind that, ultimately, the coding will be binary.

What can be held in a memory word? Well, any pattern of 16 bits. Obvious; but the point I'm driving at is that those 16 bits can mean anything we want them to mean. If we want them to mean a 2's complement coded integer then a word holds a number in the range −32768 to 32767. If we want them to mean a positive integer with no sign bit then the number is in the range Ø to 65535. If we want, we can split the word into two 8-bit fields each of which represents an alphabetic, punctuation or graphics symbol. As Tweedledee (or was it Tweedledum?)* said: "When *I* use a word, it means just what I choose it to mean—neither more nor less." I sometimes think Lewis Carroll was ahead of his time.

Now for the special-purpose registers. Just the A-register to kick off with. This is used every time you do any arithmetic. The result of any sum you ask the machine to do is put into the A-register. (Sometimes it's called the *accumulator*, by the way.) Most arithmetic operations work on two values; it's no good asking the machine to work out 3+, you need to say what 3 is to be added to. One of these values must be in the A-register before the addition operation is executed. So you can write an instruction like:

ADD (1A3)

and the machine takes that to mean:

1. Add the contents of memory location 1A3 to the contents of the A-register. (The brackets round 1A3 are being used to indicate that it's the *contents* of 1A3 and not the *number* Ø1A3 which is to be added.)
2. Put the result back in the A-register.

We've just written our first machine level instruction. It's not actually in machine code, but it's close. Look at its general form. It consists of an operation code, ADD, and an address, (1A3). Many instructions will look like that. Incidentally, life is too short to say "operation code" too often; everybody shortens it to *opcode*.

AN ADDITION PROGRAM

Let's think about a sequence of machine instructions which would model the BASIC statement:

LET R = B + C

First we would have to assign actual addresses to R, B and C. Suppose that these are 1Ø3, 1Ø4 and 1Ø5, respectively. We have to get the contents of 1Ø4 into the A-register. Let's invent an LD (for load accumulator) instruction to do this:

LD (1Ø4)

then add on the contents of 1Ø5

ADD (1Ø5)

and finally we need a way of storing the A-register's contents back in 1Ø3. So we'll invent a "store" instruction:

ST (1Ø3)

Now we have a simple machine level program consisting of 3 instructions:

LD (1Ø4)	[load B into A-register]
ADD (1Ø5)	[add on C]
ST (1Ø3)	[put the result in R]

How do we get the machine to run such a program?

* See page 152.

We're used to the idea that a program is stored in the machine *before* it's executed. After all, if you wrote the BASIC statement:

```
10 PRINT "HELLO WORLD"
```

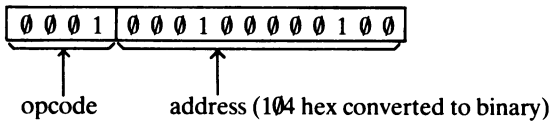
you'd be somewhat disconcerted if, as soon as you hit ENTER, the message "HELLO WORLD" were displayed. You expect it to be held until you need it. So, by the same token, a machine level program has to be stored first. Where more natural to store an instruction than in a memory word? (A word means what you want it to mean—remember?) Of course, that implies that the opcodes LD, ADD and so on have to be coded as bit patterns, but all we have to do is invent a table of bit patterns in a quite arbitrary way like this:

Opcode mnemonic	Binary code
ADD	0000
LD	0001
ST	0010

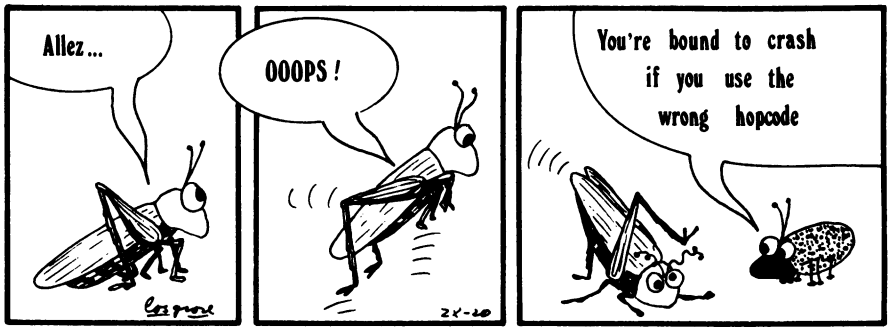
and every time we think of a new opcode that's needed, we add it to the table.

I've assumed, above, that all opcodes have a 4-bit binary code. That allows 16 different patterns and therefore 16 distinct instructions. This is a small instruction set by modern standards but it will do for our hypothetical toy computer. We've got 16 bits in the word altogether, so 12 are left for the address portion of the instruction.

So LD (104), once inside the machine looks like:

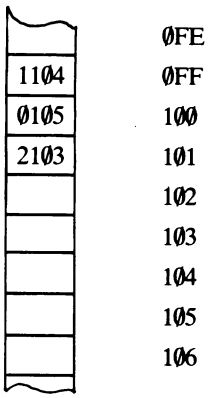


Once you've seen one bit pattern, you've seen them all, so from now on we'll write the hex versions of instructions. It's marginally less tedious.



THE PROGRAM COUNTER

Suppose we store our 3-instruction program from location ØFF onwards:



Now we need a way of saying to the machine: "Kick things off by executing the instruction in ØFF, then do the one in 1ØØ, then one in 1Ø1." That's what the PC-register, or *program counter*, is for. It acts as a kind of bookmark for the computer. We run the program by initializing the PC to the address of the first instruction. While the machine is obeying this instruction, the PC is automatically updated by 1, so that when the system returns to examine the PC, it will go and obey the next instruction, and so on.

There's a snag, though. While the last instruction (in 1Ø1) is being dealt with, the PC will be updated by 1 as usual, and so when the machine looks at it again, it will find 1Ø2, and leap off to execute the instruction there. What instruction? We didn't put one in 1Ø2. Ah! But there has to be a bit-pattern in 1Ø2 left by a previous program, or just set up when the machine was switched on. So the machine will interpret this pattern as if it is an instruction, because that's what we've asked it to do. And then it will roll on through locations 1Ø3, 1Ø4 and 1Ø5 and that's where we're storing data! So if the number in 1Ø4 is 2ØFF, for instance, the machine will interpret this as:

ST (ØFF)

which will copy the contents of the A-register into ØFF, thereby destroying the first instruction of our program! Obviously what we need is a "halt" instruction (I'll use the mnemonic HLT) which stops the updating of the PC in its tracks. So the program now reads:

LD (1Ø4)
ADD (1Ø5)
ST (1Ø3)
HLT

There's an important point here. Precisely because we are using words to mean different things at different times, we have to keep a very careful eye on the implications the machine will draw from what we tell it to do. If we request it to ADD the contents of a location to the A-register, then it will assume that that location holds a number. It will make no tests; it cannot—any bit-pattern could represent a number. Similarly, any bit-pattern could represent an instruction, so if the PC points to a location, its contents will be executed as an instruction.

The rule is: *keep data and programs firmly apart.* If you don't you can expect to be totally mystified at regular intervals. As I've indicated, a whole program can disappear without trace while it is running!

Some more instructions: the functions of the program counter and the stack.

14 Jumps and Subroutines

So far, our instruction set looks a bit thin. We've got LD and ST, which will move things around memory, ADD, which is pretty primitive arithmetic, and we can stop things with HLT.

We'll pep up the arithmetic capability a bit by adding SUB, which will subtract the contents of a location from the A-register, but that's all we're getting. No multiply, no divide, definitely no square root.

What we really need is a set of branch instructions, equivalent to BASIC's IF ... THEN ...

JUMPS

It's going to be fairly easy to branch to an instruction out of the usual sequence; what we need to do is change the contents of the PC. So we'll use an instruction like:

```
JP 416          [jump to 416]
```

Whenever it is executed, it will put 416 in the PC. The system is "fooled" into thinking that the next instruction is in 416, and then it will go on to 417, 418 etc. until the next "jump" instruction is encountered. Of course, any address can follow the JP opcode.

This instruction is more like a GOTO than an IF ... THEN .... What we need is an instruction which resets the PC only if some condition is met. The simplest test we can make is whether the A-register contains zero.

```
JPZ 2A7         [jump to 2A7 only if A-reg. contains Ø]
```

Another would be:

```
JPN 14E         [jump to 14E only if contents of A-reg. are negative]
```

That's the minimum we can get away with, because we can now test for a positive (non-zero) number by noticing when the program doesn't jump on either JPZ or JPN instructions.

SUBROUTINES AND STACKS

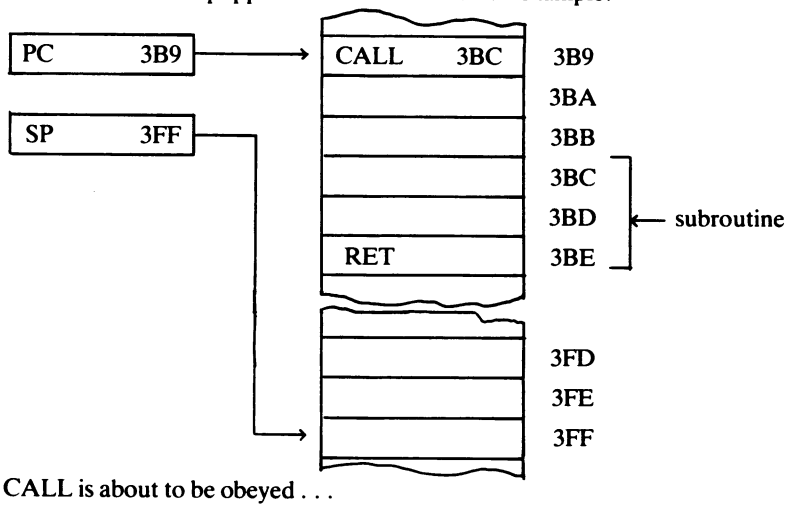
While we're on the subject of transferring control from one place to another inside the program, how about something like BASIC's GOSUB and RETURN?

We'll have an instruction:

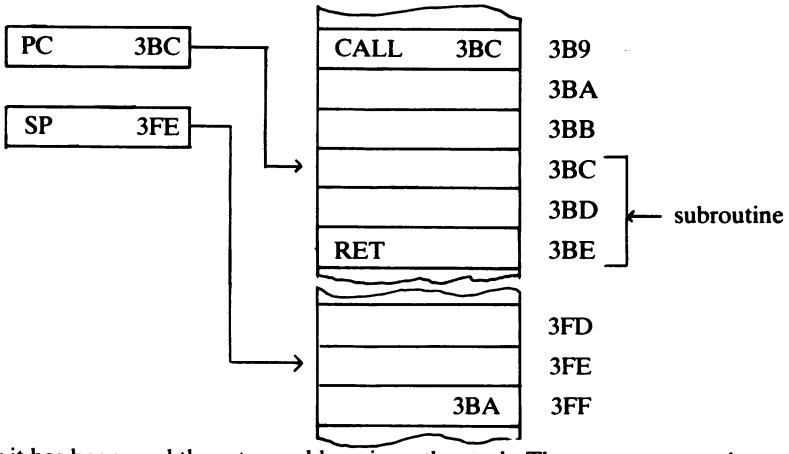
```
CALL 2Ø5        [call the subroutine starting in 2Ø5]
```

What does it do? Well, obviously it puts 2Ø5 into the PC, but we could use a JP for that. CALL performs a second function: it stores the address of the instruction after the CALL, so that when a "return" (opcode: RET) is encountered it can load the stored address back into the PC to continue the main program from where it left off. This is

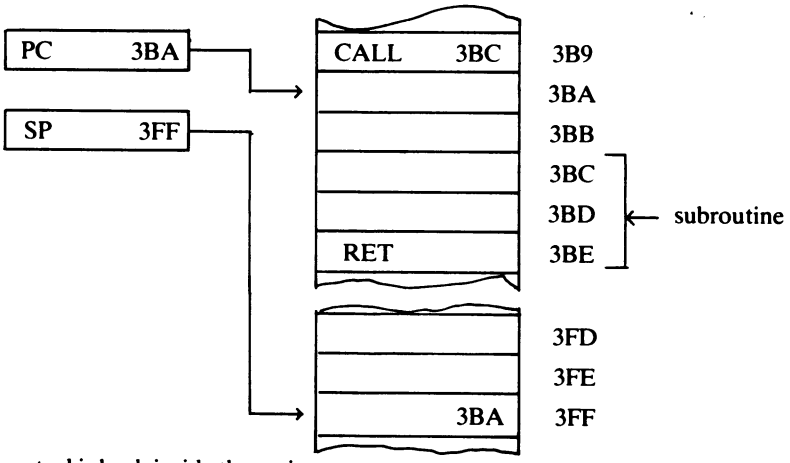
where the SP register comes in. We use some of the memory as a *stack* (remember stacks?) and SP points to the top of the stack. When a CALL is obeyed, the return address (the address of the CALL + 1) is pushed on to the stack. When the RET is encountered the stack is popped into the PC. Here's an example:



The CALL is about to be obeyed . . .



Now it has been, and the return address is on the stack. The program steps through the subroutine until it reaches the RET, after which:



and control is back inside the main program.

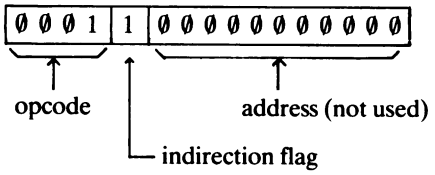
The private eye tracks his victim: how to use the contents of one address to point to another one.

15 Indirection and Indexing

There are only two registers left to talk about, and both have similar functions: they can both alter the address part of an instruction while the program is running.

INDIRECTION

Let's look at the way the I-register does this first. We'll invent a new opcode, LDI or "load indirect". Like HLT, it doesn't have an address associated with it. To the machine, it's just like an LD except that the high bit of the address field is set to "1". This bit is called the *indirection flag*, and simply indicates to the machine that indirection is in force. So the binary form of the LDI instruction is:



The hex code is 18Ø0. When the machine encounters this instruction, it uses whatever number is in the I-register as the effective address. So if the I-register contains 1E4 and an LDI instruction is executed, the effect is exactly the same as if the instruction had been LD 1E4. In other words the I-register acts as a memory pointer, and we can move it around to our heart's content if we can do arithmetic with it. That means moving values into the A-register, because that's the only place we can do arithmetic. So we'll invent an opcode XAI for "exchange contents of A-register with contents of I-register".

Of course, the indirection flag can be set for any instruction which has an address part. So we can have STI, JPI, ADDI etc. and in each case, the last 3 digits of the hex code will be 8Ø0.

AN EXAMPLE

Let's look at an example which uses these ideas. Suppose that we want to initialize a 1D array of length 2Ø, to hold the numbers 2, 4, 6, 8 . . . 4Ø. In other words we want a machine code equivalent of the BASIC:

```
FOR C = 1 TO 2Ø
LET A (C) = C * 2
NEXT C
```

There are a series of values which are going to have to be in memory somewhere, to make this work. They are 1 (because the loop count goes up in ones), 2 (because that's the increment for the array contents) and 20 (which is needed to test for the end of the loop). I don't, for the moment, want to be bothered with exactly where these numbers should be stored, so I'm going to allow addresses to be referred to temporarily by names (just like BASIC names). We'll have to convert these to numbers when we finally get to machine code, of course. This is an application of Jones's First Law of Computing: "Never put off till tomorrow what you can put off till the day after." So we'll assume that the numbers we want are available in locations called N1, N2 and N20. Similarly, we'll have a location called BASE which holds the address of the first element of the array, and one called COUNT which will act as the loop counter.

First we set the I-register to point to the base of the array:

```
LD   BASE
XAI
```

Then we set the COUNT to 1:

```
LD   N1
ST   COUNT
```

Now we double this (by adding it back into the A-register) and store it in the location pointed at by the I-register. (We talk about "storing *through* the I-register" for short.)

```
ADD   COUNT
STI
```

We "undouble" the value on the A-register again, subtract 20 and see if the result is zero. If it is we've finished:

```
SUB   COUNT
SUB   N20
JPZ   OUT
```

OUT is another, as yet unspecified, address. We don't know where it is yet, because we don't know where the program ends, and so, again, it's useful to give it a name temporarily.

If the branch doesn't occur, we add 1 to the COUNT:

```
LD      COUNT
ADD     N1
ST      COUNT
```

and increment the I-register by 1:

```
XAI
ADD   N1
XAI
```

The current COUNT is now back in the A-register, so we can loop back to the doubling operation:

```
JP      LOOP
```

provided we give the "ADD COUNT" instruction the symbolic address "LOOP". Let's do this by preceding the instruction by its symbolic address followed by a colon:

LOOP: ADD COUNT

We can do the same sort of thing to set up the initial values we need, by defining a new opcode HEX which just sets a word to a required value. It isn't really an opcode at all since it isn't equivalent to a machine instruction, so we call it a pseudo-operation. The whole program looks like this (ignore the numbers in the left- and right-hand margins for the moment):

```
020          LD      BASE      1 033
021          XAI               A 000
022          LD      N1        1 030
023          ST      COUNT     2 032
024 LOOP:    ADD     COUNT     0 032
025          STI               2 800
026          SUB     COUNT     4 032
027          SUB     N20       4 031
028          JPZ     OUT       6 047
029          LD      COUNT     1 032
02A          ADD     N1        0 030
02B          ST      COUNT     2 032
02C          XAI               A 000
02D          ADD     N1        0 030
02E          XAI               A 000
02F          JP      LOOP      5 024
030 N1:      HEX     0001      0 001
031 N20:     HEX     0014      0 014
032 COUNT:   HEX     0000      0 000
033 BASE     HEX     0000      0 000
```

The only symbolic address which doesn't appear in the left-hand column, and is therefore still unspecified, is OUT. We'll worry about it later.

The form of the program we now have is written in what is known as *assembly code*. On modern sophisticated computers there will be an *assembler program* whose function is to convert this into real machine code for us.

HAND ASSEMBLY

Alas, neither our hypothetical machine nor the ***1000*** has such a program. So we have to do the job by hand. We need a table of opcodes and their equivalent hex values:

Opcode	Hex
ADD	Ø
LD	1
ST	2
HLT	3
SUB	4
JP	5
JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

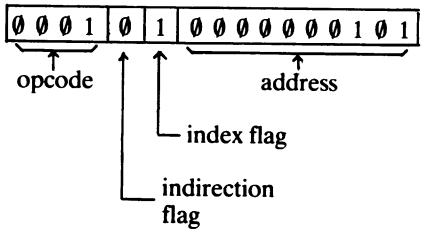
Also we need to know where the beginning of the program is. That's a more or less arbitrary decision, so let's assume it's at ØØØ. Since each instruction occupies 1 word, we can write down the address of each instruction. You'll see that I've done this down the left-hand side of the program. Now we can replace the opcodes and addresses by their hex equivalents. For instance, LD BASE becomes 1 Ø33, since BASE is now identified as Ø33. The right-hand margin shows the complete code.

The only instruction which needs further comment is JPZ OUT, which encodes as 6 Ø47. Why should OUT be at Ø47? It could be elsewhere, but Ø47 is the first location it can be at. The reason is that the array is occupying the space from Ø33 to Ø46 (twenty words), and we obviously don't want to go clumping around inside the program's data area.

THE INDEX REGISTER

When the X-register is in use, the real instruction address is formed by adding the address field to the contents of the X-register. For instance, if the X-register contains 4ØØ, then the instruction LDX ØØ5 has the same effect as LD 4Ø5.

We'll pinch another bit of the address field to indicate when indexing is in operation, so the LDX ØØ5 instruction looks like this:



In hex, that's 14Ø5.

Actually there's nothing you can do with indexing that you can't do with indirection. It's just that it will do arithmetic with addresses automatically instead of leaving the job to you.

The actual architecture of the Z80 CPU, the heart (or is it the brain?) of your ZX81, T/S 1000 or Spectrum

16 At last the Z80!

I'm sorry that you've had to wade through the last ten or so pages without being able to try anything out, but if you've really understood the ideas in them, you'll find that understanding the Z80 is a breeze.

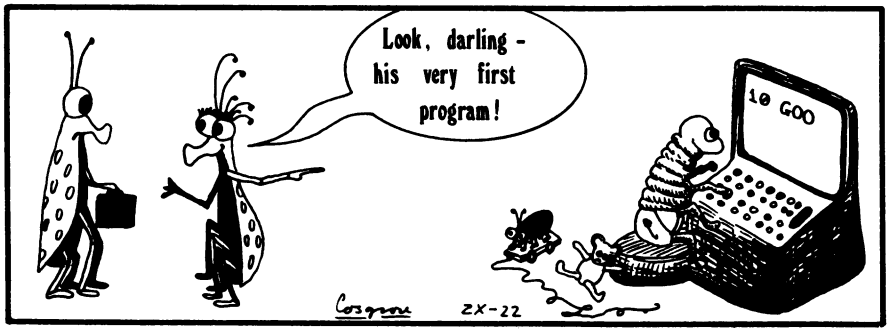
Before we get into the Z80's architecture (sorry, the chapter heading isn't quite accurate!) let's consider some of the difficulties of the processor I've just described.

First, the 4-bit operation code only allows 16 different instructions. (OK, we cheated a little, by allowing the indirection and indexing flags to spill over into the address field, but that in turn means we've limited the address size, and therefore the maximum size of memory!) The Z80 has 694 instructions! To give each of them a separate bit pattern means that we need an 8-bit field (1 byte); and even then some fudging is needed.

Second, our imaginary machine uses memory in a rather careless way. Some of the instructions don't use the address field (HLT, LDI, STI, for instance), so a sequence of such instructions wastes 10 bits in every word. The Z80 gets over this problem by allowing different instructions to have different lengths. Some instructions have no address field and are just 1 byte long. Others have a 1-byte address field and so are 2 bytes long. Others have a 2-byte address field for a total of 3 bytes. There are even some which have 2-byte opcodes! This means that the PC can't increment by 1 for every instruction executed. It has to increment by the length of the instruction.

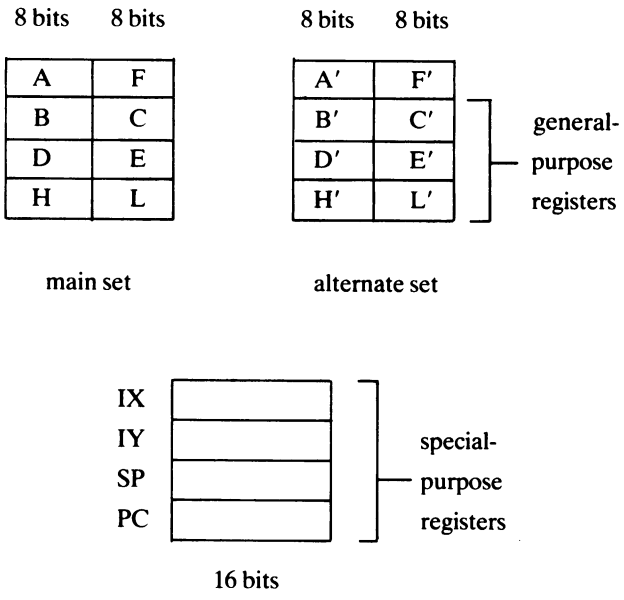
Third, we always have to handle 16-bit words, which is inconvenient if we're dealing with characters (which normally occupy a byte each). So it would be nice to allow 8-bit *and* 16-bit operations.

Fourth, the fact that there is only one general-purpose register (the A-register) can be annoying. It often means that intermediate results have to be stored temporarily back in memory while some other calculation is done. The Z80 has a number of general-purpose registers; although, as we shall see, exactly how many there are varies depending on what we're using them for.



THE REGISTERS

Here's the register organization:



Ignore the alternate set for the moment.

The registers appear in pairs, indicating that they may be used either as 8-bit or 16-bit registers. For instance, we can refer to the B-register (8 bits), or the C-register (8 bits) or the BC register (16 bits). The B, C, D, E, H and L registers can all be used in this way (in pairs BC, DE, HL *only*) but the A and F registers are strictly 8-bit registers and cannot be combined. For the 16-bit pairs, the senior byte is the left-hand one (B, D, H) as you'd expect.

There are two index registers, the IX and IY, a stack pointer (SP) and program counter (PC). What, no indirection? Actually any of the 16-bit general-purpose register pairs (BC, DE or HL) can be used for indirection but, for simplicity, we shall always use the HL for this purpose.

There are two sets of instructions, one for handling 8-bit operations and the other for handling 16-bit operations. We'll start with the 8-bit "load" instructions.

How to shuttle data from one place to another . . .
and a few words about addressing modes.

17 Load

Let's look at the "load" (LD) operation as an example of the 8-bit group. It's very like the LD instruction in our imaginary machine, except that two extra addressing modes are allowed: *register-to-register*, and *immediate*. That gives a total of five, with *direct*, *indirect* and *indexed* available as before.

1. *Direct addressing*
   This looks much the same as our imaginary equivalent, except that, since there is more than one register, we have to specify which register we want loaded:

   LD A, (ØF1C)

   This loads the contents of ØF1C into the A-register. Note that, by convention, the movement is from right to left, so that we can write:

   LD (ØF1C), A

   and mean "copy the contents of the A-register into ØF1C".
   (Actually, the A-register is the only *8-bit* register which can be directly addressed.)

2. *Indirect addressing*
   Again, this is straightforward. Since we're going to standardize on the HL for indirection, the instruction format is:

   LD A, (HL)

   which means "load the A-register *through* (i.e. from the address contained in) the HL register". To pass data in the opposite direction we could have:

   LD (HL), A

   which puts the contents of A into the address *contained in* HL. (For this instruction, registers other than A are allowed.)

3. *Indexed addressing*
   Here, we need to indicate which index register is in use, and the amount of the offset:

   LD A, (IX + 2E)

   Note that in direct addressing, I showed an address of 4 hex digits, because 16 bits (2 bytes) are allowed for the address. The offset value in an indexed address instruction must be held in 1 byte, however, so I've only shown two hex digits.

4. *Register-to register*
   We can transfer data between registers like this:

   LD D, B

   which means: "load the contents of B into D".

5. *Immediate*
   Here, data itself, rather than the address of data, is placed in the address field. So we can write:

   LD B, Ø7

   to mean "put the number 7 in B". Note again that the number is two hex digits, since it has to be stored in the single byte of the B-register. Note also that a "LD" is really a *copy*: the numbers are retained in their original addresses or registers, but a copy is placed at the destination.

HEX CODES

Now let's see what each of these instructions looks like in hex; for a full listing see Appendix 5.

1. LD A, (ØF1C)

First we look up the opcode for the LD A, (nn) instruction. (The nn indicates a general 2 byte address). This is 3A. So you would expect the instruction to code as:

3A ØF 1C

Unfortunately, there's a slight complication caused by the way the Z80 thinks about numbers; it likes the least significant (junior) byte of an address first. So we have to *swap the address bytes round*:

3A 1C ØF

This is mildly annoying, but you soon get used to it. It is an invariable rule for 2-byte numbers in Z80 instructions: *junior byte first, then senior*: hence all those PEEK X + 256 * PEEK (X + 1)'s in the Timex *Manual*.
The LD (nn), A instruction has the code 32, so:

LD (ØF1C) becomes 32 1C ØF

2. LD A, (HL)

This is easy. There is no address part so it's just a 1-byte opcode. Look it up and you'll find it's 7E.
Similarly LD (HL), A codes as 77.

3. LD A, (IX + 2E)

The general instruction is LD A, (IX + d), d indicating a 1-byte displacement (in 2's complement notation), and its code is DD 7E. (Note that—a 2-byte opcode!) So the instruction is:

DD 7E 2E

where the byte 2E is the displacement chosen in this case.

4. LD D, B

No problem here, again. The code is 5Ø.

5. LD B, Ø7

The opcode is Ø6 so the instruction is Ø6 Ø7.

To kick off with, here's a machine code program to add two numbers—and a simple BASIC loader to write and run machine code easily.

18 Arithmetic

What about arithmetic? There's an ADD and a SUB instruction, both of which reference the A-register, and which may use any of the addressing modes except direct.

So let's try writing a program to add the numbers 4 and 7 together. This would work:

```
LD A, Ø4        [put 4 in the A-reg.]
LD B, Ø7        [put 7 in the B-reg.]
ADD A, B        [add them, and put the result in the A-reg.]
```

Now store the result away somewhere:

```
LD (43ØØ), A
```

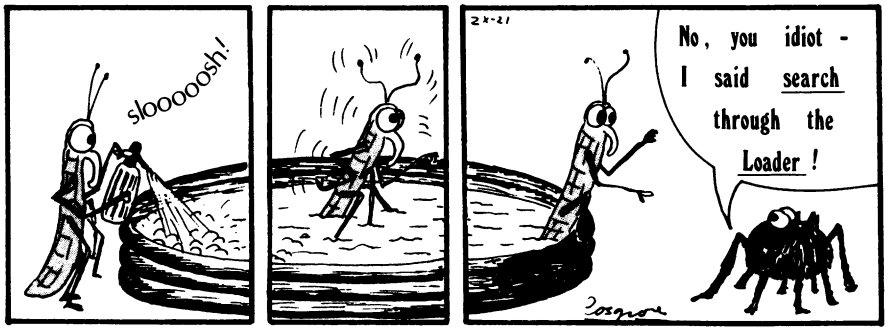
Here's the program, the hex code, and the decimal equivalent:

Program	Hex	Decimal
LD A, Ø4	3E Ø4	62 Ø4
LD B, Ø7	Ø6 Ø7	Ø6 Ø7
ADD A, B	8Ø	128
LD (43ØØ), A	32 ØØ 43	5Ø ØØ 67

LOADER

We're left with the problem of loading this code into the ***1ØØØ***, and then executing it. Since we're going to do a number of machine code routines, it's going to be worthwhile writing a BASIC program which loads and then executes machine code.

This is fairly easy. In principle, all we need to do is to ask the user where he wants to put the code in memory, then ask for each byte of code in turn, and POKE it into the appropriate location. Then we run the program by calling the USR function. Finally, we PEEK all the program locations and data area to ensure that the program is still intact (remember, it's possible to overwrite a program by accident) and that the results are correct. Obviously, it makes sense to have the data and program areas adjoining. So we'll adopt this convention: the data area always precedes the program area, and is loaded with zeros to start with. So we'll begin by asking the user the size of his data area (as a number of bytes).



There's one other problem; according to the Timex *Manual* all routines called by USR have to end the same way:

LD A, 1E	3E 1E	62 30
LD I, A	ED 47	237 71
LD IY, 4000	FD 21 00 40	253 33 00 64
RET	C9	201

(Actually it's that final RET that is crucial.) We call this the *standard ending*. So we might as well make the program generate this code at the end of the routine automatically. Here's the loader in its simplest form. (In 1K you can save memory by shortening the PRINTed phrases.)

```
10 PRINT "BASE ADDRESS:□";
20 INPUT B
30 PRINT B
40 PRINT "NO. OF DATA BYTES:□";
50 INPUT D
60 PRINT D
70 FOR I = 0 TO D - 1
80 POKE B + I, 0
90 NEXT I
100 LET A = B + D
110 PRINT "CODE:"
120 INPUT C
130 IF C < 0 THEN GOTO 180
140 PRINT C
150 POKE A, C
160 LET A = A + 1
170 GOTO 120
180 CLS
190 FOR I = 1 TO 9
200 POKE I - 1 + A, M (I)
210 NEXT I
```

The last three lines assume that the array M has been set up in command mode (i.e. without line numbers) by:

```
DIM M (9)
LET M (1) = 62
LET M (2) = 30
LET M (3) = 237
LET M (4) = 71
LET M (5) = 253
LET M (6) = 33
LET M (7) = 0
LET M (8) = 64
LET M (9) = 201
```

(These are the standard "end of routine" codes mentioned above).

Or, you could input these values into M using a FOR loop, and then delete the loop before saving. It's probably just as longwinded, though. Either way, don't forget to execute the program with GOTO 10 and not RUN, so that the array values are preserved.

Now we must execute the program:

```
220  LET Y = USR (B + D)
```

(Don't forget that B + D is where the program begins. Before that there's just data). The value, here Y, returned by USR isn't usually needed, but it has to be there to satisfy the syntax of the statement. It actually contains whatever was in the BC register pair on returning from the machine code routine.

Finally, we look at the state of the program and its data:

```
230  FOR I = B TO A + 8
240  PRINT I, PEEK I
250  NEXT I
```

RUNNING

Now, to run the program. Firstly, the machine code routine has to coexist with the BASIC system. If we're careless, BASIC will clobber our defenceless little routine the first chance it gets, because it's always moving things around in memory in ways that aren't always easy to predict. One way out of this problem is to fool BASIC into thinking that the top of memory is below where it really is, and use the resulting "attic" for any programs we want preserved. To do this, we POKE the bytes 16388 and 16389 (which together form a system variable called RAMTOP) with the address from which we wish to start our program. In other words, this is the first address which is unavailable to BASIC. As usual, the low byte contains the least significant value. So, taking an example for a 1K machine, RAMTOP contains the hex value 4400 to start with. If we want to allocate a 256 (decimal) byte attic, we have to set RAMTOP to 4300 so:

```
POKE 16388, 0          [= 00 hex]
POKE 16389, 67         [= 43 hex]
```

(Incidentally, 4300 hex = 17152 decimal, and you can omit the POKE 16388 unless, for some reason, you've changed it from its usual 0 value previously.)

Now we type NEW because BASIC only notices that RAMTOP has changed when NEW is executed. Next, load the "loader" program and run it. In response to its BASE ADDRESS request, type 17152, and, to NO. OF DATA BYTES, type 1. Finally, key in the machine code (62, 4, 6, 7 etc.) terminating with a negative value, a delimiter (see ***Timex/Sinclair 1000***, p. 39) ignored on loading but signalling "end of code listing".

The system responds by printing the contents of bytes from 17152 onwards. In 17152 is 11, which is the sum of 4 and 7, and this shouldn't surprise us much, since that's where we asked to store the result, and it's also the byte we allocated for data. The rest of the "memory dump" just confirms that the program is correctly stored.

Experiment, by altering the values being added. (Just POKE new values into 17154 and 17156 and GOTO 22Ø). Or put the result somewhere else—17153, say. See how it changes the program?

In particular, try adding 24Ø to 1ØØ (decimal). The result isn't 34Ø! Why?

Think about it in binary:

```
24Ø        1 1 1 1 Ø Ø Ø Ø
1ØØ      + Ø 1 1 Ø Ø 1 Ø Ø
           ---------------
           Ø 1 Ø 1 Ø 1 Ø Ø  =  84
           ---------------
           1 1
1
```

The sum generates a 1 in the ninth bit, which can't be held in an 8-bit byte, so it falls off the end and the quoted result is too small by the value of that ninth bit—256. No check has been made, no helpful error message printed. When you write machine code you're on your own. What you don't test for, you don't find out about.

AN IMPROVED LOADER

For that first try, I gave you a program that loaded *decimal* opcodes. That was so you could concentrate on the mechanics without worrying about hex codes. But hex is so much more convenient. Here's how to modify LOADER to accept hex, by combining it with the decimal/hex converter in Chapter 11.

Change lines 12Ø, 13Ø, 14Ø, 15Ø to:

```
12Ø  INPUT C$
13Ø  IF C$ = "S" THEN GOTO 18Ø
14Ø  PRINT C$
15Ø  POKE A, 16 * (CODE C$ (1) - 28) + CODE C$ (2) - 28
```

The procedure is exactly as before; but now at each input you key in the hex code: 3E, then Ø4, then Ø6, etc. Don't omit the zeros. Use "S" to end the inputs, in place of the previous "negative number" delimiter.

In the above description, I assumed you might only have 1K of memory. Machine code is of course a useful space-saver with 1K; but anyone interested in it is likely to have 16K. Further, some of our later routines using the display file need at least 4K. To reserve a 256-byte attic in 16K, proceed as follows.

```
POKE 16389, 127
NEW
```

Now load in the machine code. Replace 4300 hex by 7F00 hex; and 17152 decimal by 32512 decimal. Appendix 2 contains a table for allocating memory in 256-byte blocks in 16K.

From now on, I'm only going to give the hex codes. You can either modify LOADER as above, or you can convert from hex to decimal using Appendix 1. Since the latter is tedious, and it's easy to make errors, I strongly recommend the former.

A really versatile loader is given in Appendix 6: HELPA.

A USEFUL TRICK

It's a terrible nuisance that the ***1000*** needs a NEW before it recognizes changes to RAMTOP. You're half way through typing in a BASIC program, with some machine code to accompany it later . . . Bother! You've forgotten to allocate memory.

Using what I've told you so far, all you can do is SAVE on tape; reset RAMTOP, then LOAD back in, and continue. But there *is* a way to avoid this, by using a ROM routine. (As I say elsewhere, I don't think that learning a list of ROM routine addresses to call is really the way to get into machine code; but this is so useful, I'll make an exception.)

Suppose you want to allocate a 256-byte attic, leaving your precious BASIC intact. From the keyboard, type

```
POKE 16389, 127          [reserve space]
PRINT USR 1040           [call from ROM]
```

You'll get a listing and the program will halt. Restart with RUN (this technique loses your variables anyway). That's it. For different sizes of attic, change the 127 (and POKE 16388 with the junior byte if this isn't zero).

You can use this in a program, but the program will halt after the USR 1040, and need a manual restart. And directly entered variables will be lost.

You can store machine code other places than above RAMTOP; see Chapter 22. But all methods have their quirks.

There are 694 Z80 instructions: here's a selection of the more fundamental and accessible ones, and what they will do.

19 A Subset of Z80 instructions

I'm not going to describe every one of the 694 opcodes the Z80 has; that would be tedious and unnecessary. (But see Appendix 4.) We'll look at a subset of 30-odd types of instruction (covering about 230 actual commands). Unfortunately, not all of them can use all the addressing modes. Here's a quick reference table showing which instructions can use what; the opcodes are given in Appendix 5.

Address Mode \ opcode	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Register	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Immediate	LD r, n	ADD A, n			JP nn	JPZ nn	LD r, nn		
Direct	LDA, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indirect	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Indexed	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					

8-bit operations | 16-bit operations

The notation in the table needs some explanation. Some of the opcodes will be unfamiliar, but we'll deal with those later. Otherwise, the conventions are:

1. Each entry in the table shows an example of the format of the instruction type. Any of the other opcodes in that column could be substituted.
2. "r" or "s" denotes any register. Whether this is an 8-bit or a 16-bit register depends on which part of the table the instruction is in. For instance, in the LD r, s instruction, r and s are any 8-bit registers (A, B, C, D, E, H or L), but in ADD HL, r "r" is one of BC, DE, HL, SP.
3. "n" is any 8-bit number. "nn" is any 16-bit number.
4. If a register is explicitly stated, as in LD A, (nn), then this is the only register which may be used for this purpose.

   This is a wild oversimplification. Sometimes, other registers are usable, but the point is that the set of instructions I've shown are always OK and you can worry about extending your vocabulary of instructions when you're handling this lot confidently.

5. "d" is any 8-bit number, but it's always added to some 16-bit value. In other words, it's an indexing displacement.

Now let's look at the new opcodes:

AND

This operation takes the contents of the A-register, and another 8-bit field, and examines these, bit by bit. Only if corresponding bits are both "1" does it put a "1" back in this position in the A-register. Otherwise it inserts a "Ø".

For instance, AND A, Ø7 has the following effect:

A-register before the operation:	Ø Ø 1 1 Ø 1 Ø 1
Ø7:	Ø Ø Ø Ø Ø 1 1 1
A-register after the AND:	Ø Ø Ø Ø Ø 1 Ø 1

See how the junior three bits have been transmitted? So you can use AND to select a portion of a byte.

OR

This works in a similar way to AND, but this time, the resulting bit is a "1" if either of the initial bits is a "1". So OR A, Ø5 gives:

A-register before:	Ø 1 Ø Ø 1 Ø 1 1
Ø5:	Ø Ø Ø Ø Ø 1 Ø 1
A-register after:	Ø 1 Ø Ø 1 1 1 1

Now, certain bits are being forced to "1" regardless of their original value.

XOR

Here the initial bit values must be different for the result to be a "1". XOR A, B3 gives:

A-register before:	Ø 1 Ø 1 1 Ø 1 Ø
B3:	1 Ø 1 1 Ø Ø 1 1
A-register after:	1 1 1 Ø 1 Ø Ø 1

It's particularly useful for flipping a register from Ø to 1 and back again. If the A-register contains Ø to start with, every time the instruction XOR A, Ø1 is executed, the value in the A-register will flip. (Ø to 1, back to Ø, back to 1 and so on.)

CP

This is the "Compare" instruction. The contents of the A-register are compared with those of another 8-bit field. That raises a problem, though: how is the result of the comparison signalled?

This is what the F (or *flags*) register is used for. Each bit of the F-register holds some information about the effect of the last instruction to alter them. (Not all instructions do alter them).

The flags which most interest us are the Carry, Zero, Overflow and Sign flags. CP can alter any of these, but the one of most significance here is the Zero flag, which is set if the two values being compared are equal.

If the A-register contents are *less* than those of the compared byte, the sign flag is set. This is equivalent to saying "the result is negative". This is all you need to know about the flags at the moment: it's an intricate topic if you delve deeper.

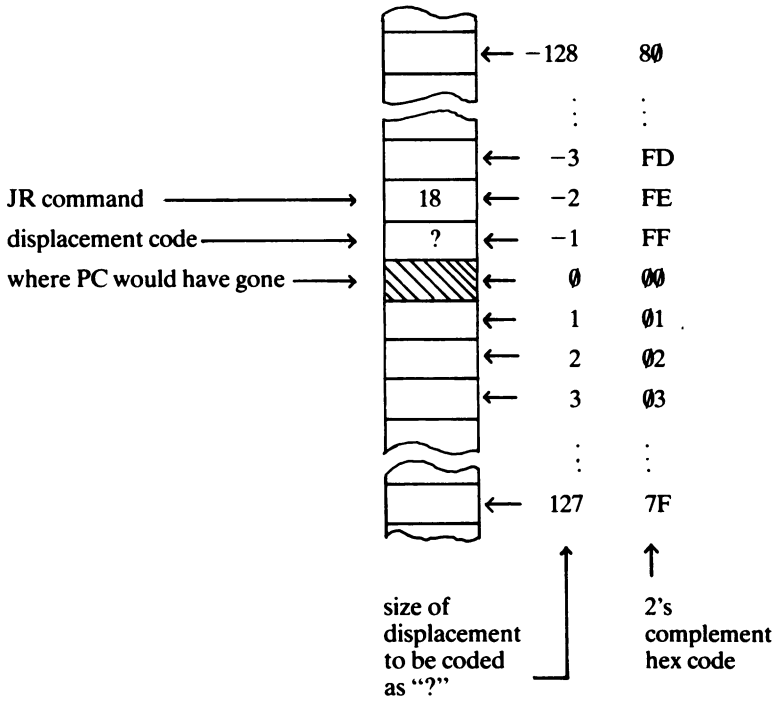
THE JUMPS

All the conditional jumps branch (or not) depending on the contents of the flags. So, for instance, JPZ says "jump if the Zero flag is set". Now we can see how the CP instruction can be used. Suppose, for example, that we wish to see if a particular byte, pointed at by HL, contains 1E hex. If it does, we want to branch to 447B. The code is:

LD A, 1E	3E 1E
CP A, (HL)	BE
JPZ 447B	CA 7B 44

All the other jumps behave similarly; JPNZ says "jump on a non-zero result" (zero flag *not* set), JPP says "jump on a positive result" (sign flag *not* set), JPM says "jump on a minus result" (sign flag set), JPNC says "jump on no carry" (carry flag *not* set), and so on. All of them have one thing in common, and that is that the address of the jump is fixed. In other words, if, for any reason, we would like a routine to run somewhere in memory other than where we first loaded it, all the jump addresses must be changed. The Z80 deals with this neatly by allowing "relative jumps" (JR). In other words, you can jump so many bytes forward (or back) from where you are. This displacement is held (in 2's complement notation, Appendix 1) in 1 byte, so the distance which can be jumped can't exceed 128 bytes backwards or 127 bytes forwards.

The displacement is calculated from what the PC value *would* have gone to next, had no jump occurred; namely, the address of the next command in the program. Like this:



Here's an example. We want to examine each byte of memory in turn for the first occurrence of 1E hex. Assume for simplicity that the start address is already in HL. We could write:

```
        LD A, 1E
LOOP:   CP A, (HL)
        INC HL
        JRNZ LOOP
```

Two points need explaining. First, I've sneaked in a new instruction: INC. This is short for INCrement. It just adds 1 to the contents of the specified register; so the compare operation is always looking at the next memory byte because HL is being bumped up by 1 every loop. (By the way, DEC, short for DECrement, does exactly the opposite.) The second point is that there's no obvious difference between JRNZ LOOP and JPNZ LOOP. It isn't until we *assemble* the instructions into machine code that the difference is clear. Suppose the code is to be loaded from 4300 hex:

Address		Instruction	Hex code
4300		LD A, 1E	3E 1E
4302	LOOP:	CP A, (HL)	BE
4303		INC HL	23
4304		JRNZ LOOP	20 FC

Why is FC in the address part of the JRNZ instruction? It works like this: when the JRNZ instruction is executed the PC is bumped up by 2 because it's a 2-byte instruction. So the PC is now at 4306. We want to jump to LOOP, which is at 4302, 4 bytes back, or −4 bytes away, to use the Z80's way of thinking about it. Now, 4 in binary is 00000100 and we create −4 by flipping the bits and adding 1 (2's complement, remember?). So:

```
0 0 0 0 0 1 0 0
                    flip the bits
1 1 1 1 1 0 1 1
            + 1     add 1
---------------
1 1 1 1 1 1 0 0
                    convert to hex
    F     C
```

Another thing which may be worrying you: INC HL doesn't alter the flags, so I'm safe to test after the increment.

The same program with absolute jumps would have looked like:

Address		Instruction	Hex code
4300		LD A, 1E	3E 1E
4302	LOOP:	CP A, (HL)	BE
4303		INC HL	23
4304		JPNZ LOOP	C2 02 43

Notice that the JPNZ instruction has 3 bytes because it contains a whole 16-bit address; and don't forget about swapping the 2 bytes of that address around!

There's one very powerful instruction in the Jump group I haven't mentioned yet—DJNZ. It decrements the B-register by 1 and jumps (relative) only if the result is non-zero.

Suppose our little "search for 1E" program is only to search a region one hundred (hex 64) bytes long, after which it should leave the loop whether it's found a 1E or not:

```
        LD B, 64          06 40
        LD A, 1E          3E 1E
LOOP:   CP A, (HL)        BE
        JPZ GOTCHA        CA — — (address for GOTCHA)
        INC HL            23
        DJNZ LOOP         10 F9
```

The loop is executed one hundred times, unless a 1E is found, in which case a branch to GOTCHA occurs. In other words, DJNZ acts like a simple FOR loop in BASIC.

Note that with *all* the relative jump commands JR, JRC, JRNC, JRNZ, and JRZ, the size of jump is calculated the same way. A table of 2's complement hex codes is given in Appendix 1 for hand-coding of jumps; our utility routine HELPA in Appendix 6 will work out relative jumps automatically for you, during the writing of the code.

ADC and SBC

These are the "ADD with Carry" and "SUB with Carry" instructions. I said earlier that there is a Carry flag in the flags register. This gets set if there is a carry generated out of a register by an arithmetic instruction. The ADC instruction will act just like ADD, except that it will add 1 more in if the Carry bit has been set by a previous operation. The SBC instruction works the same way, except that it will subtract the Carry flag.

THE SHIFTS

The shift instructions, SLA, SRA and SRL, all have the effect of shifting bit-patterns around.

SLA shifts the pattern left by 1 bit, so if the B register contains:

0	0	1	0	1	1	0	0

and SLA B is executed, the result is:

0	1	0	1	1	0	0	0

(Notice that a zero is used to fill on the right.)

Since 00101100 = 44 and 01011000 = 88 (decimal) you can see that the effect is to multiply by 2.

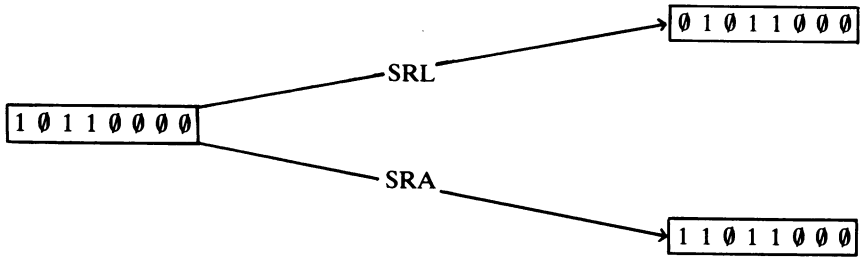
Another SLA B will give:

1	0	1	1	0	0	0	0

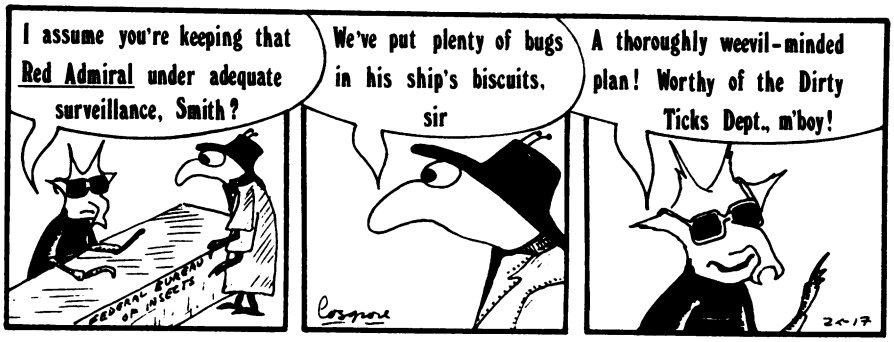
Since the senior bit is now 1, this will be seen as a negative number, and the Sign flag will be set. So far as the programmer is concerned, what's happened is that the value (176) can't be held in a byte, so we've got an overflow condition.

Right shifts work much the same way, but there's one important thing to note: SRL fills the senior bit with a zero, but SRA fills with whatever was there before.

For instance:



The reason is this: SRL is a *shift right logical*, which simply shifts the bit pattern without altering it. SRA is a *shift right arithmetic*, which treats the operation as "divide by 2". Now, when a negative number is divided by 2 the result should still be negative, so we have to preserve the sign bit.



PUSH and POP

You'll probably remember these terms from our discussion on stacks. They're used here in exactly the same way, and allow us to access the machine stack other than through a subroutine call.

This can be useful for saving values temporarily. For instance, suppose you've got a value in BC which you want later, but just now you'd like to use BC for something else. You can write:

```
PUSH BC
.....
Code
using
BC
.....
POP BC
```

This is often done before a subroutine CALL as well, so that it doesn't matter what registers the subroutine uses: it can't interfere with the calling program's data. You may see code like:

```
PUSH BC   ┐
PUSH DE   ├─ save the registers
PUSH HL   ┘
CALL 4FA1
POP HL    ┐
POP DE    ├─ restore register values
POP BC    ┘  (note the order!)
```

assuming that the A-register is manipulated by the routine, so we don't need to save it.

Warning

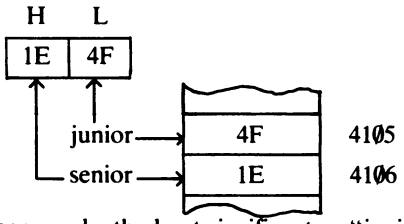
Unless you deliberately choose to alter it, the stack pointer SP will be set according to the operating system of the ZX81. There's no harm in leaving it at that value, provided you make sure that PUSHes and POPs cancel out in pairs, so that SP returns to its initial value on leaving the machine code routine. Similarly CALLs and RETs have to match. (USR generates a CALL, matched by the final RET that is tacked on to the end by the LOADER routine.)

A 16-BIT QUIRK

One feature of the 16-bit operations (PUSH, POP, LD in particular) which is important to grasp is the order in which bytes are transferred from register to memory and vice versa. It's like this:

LD (41Ø5), HL

will have the following effect, if HL contains 1E4F:



In other words, the least significant or "junior" byte in the register is loaded into the specified address; and the most significant or "senior" byte is loaded into the byte following this. Conversely,

LD HL, (41Ø5)

would have exactly the reverse effect. (NB: it codes as 2A Ø5 41, following the standard convention!) Similarly

LD HL 1ØØØ

(an attempt to load HL with the value 1ØØØ hex) encodes as

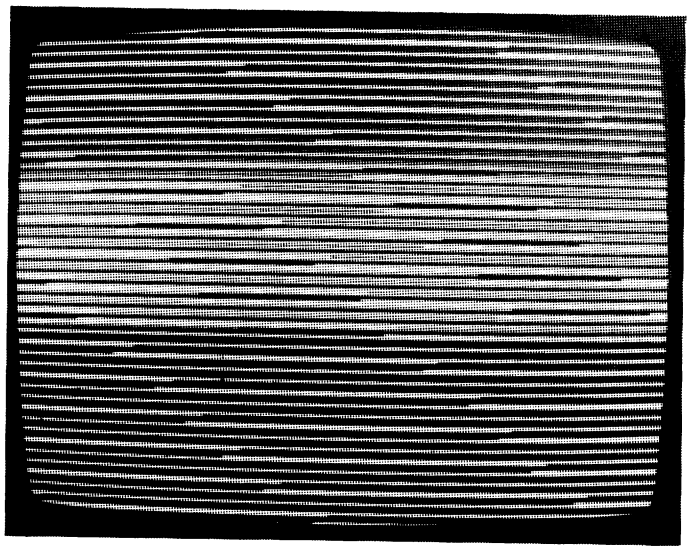
21 ØØ 1Ø

so that, even though 1ØØØ is data, not an address, its bytes get transposed as usual.

CRASHES

When a BASIC program crashes, there's little harm done: you can always break out, one way or another, without losing the program. But machine code crashes are more spectacular, and infuriating. Spectacular, because they often signal their presence by drawing op-art patterns all over the screen; and infuriating because (on the ***1000***) the only way to break out of them is to pull the power plug out and lose the contents of RAM. You want to see a crash, to check this? OK, try this little program:

```
1 REM X
2 POKE 16514, 118
3 RAND USR 16514
```



A Machine Code crash. Start again! The variety of "crash" displays will amaze you . . .

The screen blanks, and the machine no longer responds to the keyboard. This is because it uses a ROM routine to scan the keyboard, but the BASIC operating system isn't in use during a USR call of a machine code program. Once a crash occurs, you're stuck with it: pull the plug and start again. (However, there's no way to alter the ROM contents, so don't worry about doing any lasting harm! It is you, not the ***1000***, that will suffer!) But there are some simple precautions worth taking.

1. Check all machine code listings scrupulously and make sure you have input them correctly.
2. *Never* use HALT (hex code 76).
3. Make sure that CALLs and RETs match, as do PUSHes and POPs.
4. Make sure you call the correct starting address.
5. Unless there's not much to lose, SAVE what you can on tape before calling USR. (Our utility programs HELPA and DOWNLOAD permit this in useful ways.)

Machine code has no instruction for multiplying numbers;
but you can do it if you combine arithmetic, logic, and shifts.
Digging deeper now . . .

20 A Machine Code Multiplier

Now let's write a few simple routines. Remember I said that there's no Z80 multiply instruction? Let's write a subroutine to do the job.

AN EXAMPLE

First we should examine the nature of the problem, and there's no better way of doing that than looking at an example. We'll keep things as simple as possible, and work in 8-bit registers; so if we want to multiply 9 by 13 it will look like:

```
    0 0 0 0 1 0 0 1
×   0 0 0 0 1 1 0 1
```

Now we can treat this as a conventional long multiplication, but because it's in binary, it's actually easier than usual; if the current digit we're multiplying by is 1, copy the top line; if it's zero, do nothing:

```
    0 0 0 0 1 0 0 1      P
×   0 0 0 0 1 1 0 1      Q
-----------------
    0 0 0 0 1 0 0 1
    0 0 1 0 0 1 0 0
    0 1 0 0 1 0 0 0
-----------------
    0 1 1 1 0 1 0 1
```

Of course we've had to add in zeros on the right at each stage, just as we would in a decimal long multiplication. In machine code terms, that's equivalent to a shift left. I've called the two numbers P and Q, for reference.

While P is shifted left, it's also going to be convenient to shift Q right, because that way we only need to keep examining the junior bit of Q to determine whether to add P into the sum or not.

PROCEDURE

Assume that P and Q are in the D and E registers. The procedure is:

1. Set the A-register to zero.
2. If the junior bit of E is 1 then add D into A.
3. Shift D left.
4. Shift E right.

(steps 2–4: repeat these steps 8 times)

Here's a first stab at the code:

```
        LD A,Ø0
        LD B,Ø8
```

The first step's obvious; the second sets B to act as a loop counter in conjunction with a DJNZ to come at the end. Now we want to test the junior bit of E. The only way we currently have of doing that is to use a mask pattern (ØØØØØØØ1) with an AND operation, so let's set up the C register to that pattern:

```
        LD C,Ø1         [see below for hex coding]
```

We can only AND with the A-register, which will destroy its current contents, so we'll save it in L first:

```
LOOP:   LD L,A
```

then extract the junior bit of E, and restore the A-register:

```
        LD A,C
        AND A,E
        LD A,L
```

If the result of the AND was zero, we need to jump round the "add D into A" part of step 2 so:

```
        JRZ SHIFT
```

(Note that since LD doesn't affect the flags, the JRZ still refers to the AND.) Otherwise perform the ADD:

```
        ADD A,D
```

Now do the shifts:

```
SHIFT:  SLA D
        SRA E
```

and see if we've done the loop enough times yet:

```
        DJNZ LOOP
        RET
```

THE CODE

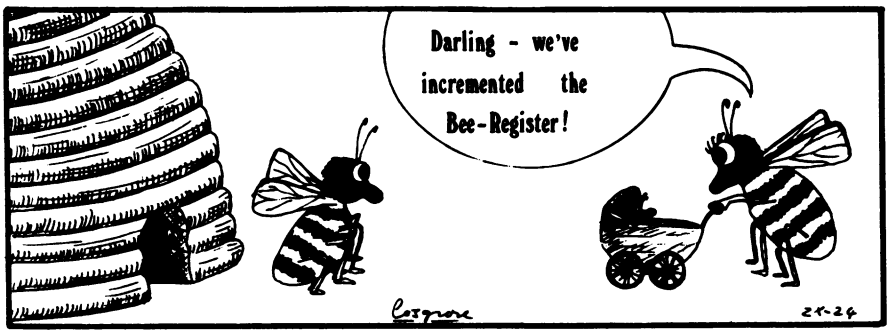
Here's the whole thing:

Address		Instruction	Hex code
0000		LD A, 00	3E 00
0002		LD B, 08	06 08
0004		LD C, 01	0E 01
0006	LOOP:	LD L, A	6F
0007		LD A, C	79
0008		AND A, E	A3
0009		LD A, L	7D
000A		JRZ SHIFT	28 01
000C		ADD A, D	82
000D	SHIFT:	SLA D	CB 22
000F		SRA E	CB 2B
0011		DJNZ LOOP	10 F3
0013		RET	C9

If you want to try this program out, you'll have to arrange for the D and E registers to hold the values to be multiplied. So you could precede the program by something like:

LD HL, 4300	21 00 43
LD D, (HL)	56
INC HL	23
LD E, (HL)	5E

and then POKE 4300 (hex) and 4301 (hex) with the values to be multiplied, before calling the program. These two bytes will, of course, be the two zero bytes at the beginning of the routine, so the LD HL, 4300 will start in 4302. Note that I didn't assign actual addresses to the program, but simply started at zero. This is because all the jumps are relative, so actual addresses are unimportant; only displacements matter. For example, with 16K you can replace all 43s in the above by 7Fs, to work with a 256-byte attic.



You'll also need to *output* the answer: at the moment it's just sitting in the A-register. A simple way to do this is to stick the answer into the display file (see next chapter for details) by adding the following code at the end, *in place of* the C9 (RET) instruction, which is there only because I said this was going to be a *sub*routine.

0013	LD HL (D-FILE)	2A 0C 40
0016	INC HL	23
0017	LD (HL) A	77

Add this at the end; add the bytes 07 and 08 at the front (or POKE them later); enter using LOADER with 2 data bytes. The letter "S" will appear at the top corner of the screen. The code for S is 56; and that's the product of the two numbers 07 and 08 you POKEd in. Of course a more elegant display routine would be nice: think about PRINT USR or read the next chapter and design one. But for a test, this method suffices.

BIT

Now, I have a confession to make; there's an easier way of testing to see if the junior bit of E contains 1. There's an instruction BIT 0, E which does the job. So:

LOOP:	LD L, A	6F
	LD A, C	79
	AND A, E	A3

becomes just:

LOOP:	BIT 0, E	CB 43

and the LD A, L has to disappear as well.

Why didn't I tell you that in the first place? Well, firstly, I promised to use only the subset of instructions in the table, a promise I've now broken. But I've made an important point in the process—that it's possible to do things satisfactorily without knowing the full instruction set.

This has been something of an academic example; I chose it because it uses several common instructions in conventional, but not necessarily obvious, ways, but I'm not suggesting that you will find a need for dozens of 8-bit integer multiplications.

To control graphics from machine code, you must know where the display is stored, and how to alter it. Change the whole screen to inverse video, or draw a line of characters, all in a flash!

21 The Display File

So now for something more clearly useful. Fed up with BASIC's ability to draw graphics lines at a snail's pace? Let's see if we can write a machine code subroutine which will draw straight lines from any point on the screen running horizontally, vertically, or diagonally.

At least, that's the target. Let's deal with the problem in easy stages. Obviously, we need to know something about how the ***1000*** handles displays before we can get anything on the screen. As you've probably seen from the *Manual*, there's an area of memory called the *display file* from which the screen display is generated. All we have to do is store character values in this region to get them displayed.

The position of the display file isn't fixed. (It depends on the program size.) There is, though, a pointer to its first byte, called D-FILE, and D-FILE has the hex address 400C, decimal 16396. So the display file starts at PEEK 16396 + 256 * PEEK 16397.

The size of the display file isn't fixed either. (At least, it isn't on 1K systems, and although it normally is on 16K systems, this isn't guaranteed; so it's safer not to make use of any feature based on such an assumption if you can avoid it.) The display file begins with a newline character, and ends every line with one as well. So, for instance, the state of the display file after the following statements:

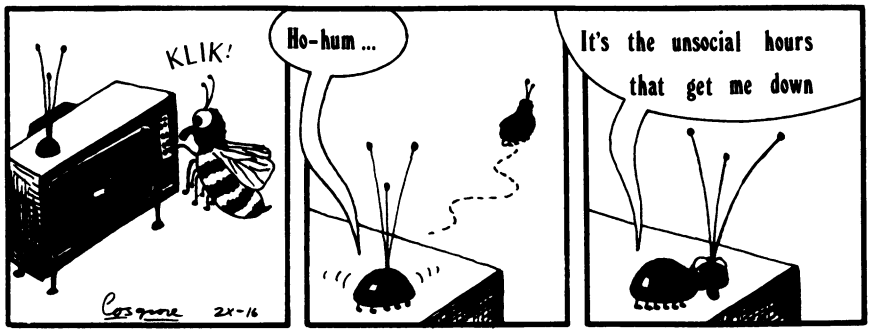
```
10 CLS
20 PRINT "ABC"
30 PRINT "□□□DEF"
```

would be:

<ABC<□□□DEF<<<<<<<<<<<<<<<<<<<<<<<<

in 1K, or in 16K if RAMTOP has been lowered to under 3¼K. With more than that size of memory, it will be padded out with the "missing" blank spaces. (I'm using "<" to represent NEWLINE and "□" to represent SPACE.)

This is a neat way of saving precious memory when only a small part of the screen is being used, but it doesn't make our problem any easier.



DISPLAYING A CHARACTER

Anyway let's try something simple, like putting a graphics symbol at the top left-hand corner of the screen. If the screen is blank to start with, there will just be 25 newlines in the display file. So our problem is to overwrite the second of them (where the "A" is in the above example) with our chosen symbol.

First, we load HL with a pointer to the display file. This is called D-FILE and is at 400C hex:

```
LD HL, (400C)          2A 0C 40
```

Now bump HL to point one character further along:

```
INC HL                 23
```

Now put the graphics character (88 hex, say) in the A-register:

```
LD A, 88               3E 88
```

and, finally, put this value where HL is pointing:

```
LD (HL), A             77
```

This works OK, but we should think carefully before extending the idea. After all, we've just overwritten one of the newlines, which is doubtless being used by one of the ZX81's system routines to handle the display. Sooner or later we'll confuse that routine if there aren't enough newlines left for it to play with. Of course, if we're overwriting a character that is already in the display, this problem doesn't arise. So, for instance, we could change the A, B or C of the first example without affecting the newlines at all. One way out of the problem, then, is to fill the portion of the screen we're interested in with spaces. On a 16K machine with RAMTOP above 3¼K, CLS does this, creating 24 lines of 32 spaces each. On a 1K machine we could write:

```
10 FOR L = 1 TO 5
20 FOR C = 1 TO 32
30 PRINT "□";
40 NEXT C
50 PRINT
60 NEXT L
```

to set up the top 5 lines to spaces.

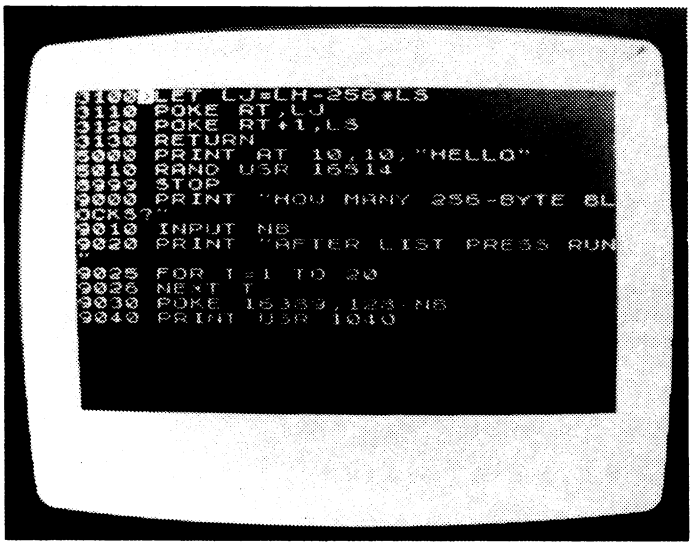
VIDEO-INVERSION

Now for something more ambitious.

Bored with the same old display all the time? How about white-on-black? We'll write a routine to invert the display file. The principle is simple: run through it changing each character to its inverse, except for NEWLINEs. For comparison, here it is in BASIC.

```
10 LET D = PEEK 16396 + 256 * PEEK 16397
20 LET B = 22
30 LET D = D + 1
40 LET P = PEEK D
50 IF P = 118 THEN GOTO 100
60 POKE D, P + 128 - 256 * (P > 127)
70 GOTO 30
```

```
100 LET B = B - 1
110 IF B = 0 THEN STOP
120 GOTO 30
```



Video-inversion of the display file.

Type this in, set up a nice display, and run the beast. Yes, well, it's a bit *slow*, isn't it? A machine code version ought to be faster. The machine code follows a very similar pattern to the BASIC program:

0000		LD B, no.-of-lines	06 16
0002		LD HL, (D-FILE)	2A 0C 40
0005	LOOP:	INC HL	23
0006		LD A, (HL)	7E
0007		CP A, newline	FE 76
0009		JRZ, SKIP	28 05
000B		ADD A, 128 dec	C6 80
000D		LD (HL), A	77
000E		JR, LOOP	18 F5
0010	SKIP:	DJNZ LOOP	10 F3
0012	(tack on standard ending via LOADER)		

The addresses are relative ones: load this up as usual with the LOADER. The relative jump sizes (underlined) have to be calculated separately at the end, using the table in Appendix 1.

Now set up a BASIC routine to print out a nice display, and call the above machine code via USR. Now it's very fast indeed! Note how the machine code mnemonics parallel the BASIC. And, a bonus in machine code: to invert video, add 128 (hex 80) to the code and don't worry about going over 256, because the extra carry digit just drops off the end of the register. (NB: with 1K, not all of the display gets inverted. Why not? Which bit does?)

LINE-DRAWING

To get a horizontal line, 10 characters long, on the top line of the display we could execute the following code:

```
        LD A, 88        3E 88       set value to be displayed
        LD B, 0A        06 0A       set loop count
        LD HL, (400C)   2A 0C 40    point to first character in display file
        INC HL          23
LOOP:   LD (HL), A      77          display
        INC HL          23          point to next character
        DJNZ LOOP       10 FC       do it again
```

To do the same job anywhere else on the display, all we need to do is alter the value in HL to start with by an appropriate offset. In principle it's easy to calculate the necessary offset. Let's think about the display file like this:

```
                                1                   2                   3
Column →    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Row  0    >□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□>
 ↓   1     □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□>
     2     □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□>
     3     □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□>
     4     □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□>
           .
           .
           .
and so on
```

If the HL is incremented after having been loaded from D-FILE so that it points at column 0, row 0, then we simply multiply the row number we want by 33 and add on the column number. That is:

offset = row * 33 + column

Provided the row value never exceeds 7, we could use our 8-bit multiplier here. But there's a neater way:

offset = row * (32 + 1) + column
= row * 32 + row + column

Despite the fact that this expression for the offset seems more complicated than the original, it has the advantage that the multiplication is now by a power of 2 (2^5), so all we have to do is shift row left 5 times to evaluate row * 32.

Now let's imagine that the row value is available in the E-register, and the column value is in the C-register. We can calculate the offset like this:

```
        LD B, Ø5        Ø6 Ø5
SHIFT:  SLA E           CB 23
        DJNZ SHIFT      1Ø FC
```

Hold on, it's not quite as easy as that! This piece of code shifts the E register contents left 5 times all right, and that's fine if row * 32 is less than 255, but it could easily be more than that, and the E-register will overflow. So we need a 16-bit register. If we use DE, the above code can be used as a basis for the routine, but there are some pieces to add on. First, we'll have to make sure that D contains zero to begin with. Second, as bits shift left off the end of E we want them to appear in D and then shift along D. This will work:

```
        LD D, ØØ        16 ØØ       clear D
        LD B, Ø5        Ø6 Ø5       load loop count into B
SHIFT:  SLA D           CB 22  ┐
        SLA E           CB 23  ┘    shift left DE
        JRNC EOL        3Ø Ø1       go to End of loop on no carry
        INC D           14          put the carry into the junior bit of D
EOL:    DJNZ SHIFT      1Ø F7       test for end of loop
```

Now we want to add this into HL, having first loaded it with the address of the first character in the display file:

```
LD HL, (4ØØC)     2A ØC 4Ø
INC HL            23
ADD HL, DE        19
```

Unfortunately, what we now need to do is to add the row value into HL, and the copy in E has been destroyed by the shift operations. That's no real problem, because we presumably passed the row value from BASIC by POKEing it to a byte just before the beginning of the machine code routine in the usual way, and it's still available there. So all we have to do is zero D, load E from this byte and ADD HL, DE again. But this does prompt the question, "Was there a neater order in which to do things?"

Well, there was:

```
        LD HL, (4ØØC)   2A ØC 4Ø  ┐  compute address of first character
        INC HL          23        ┘  in display file
        LD D, ØØ        16 ØØ     ┐
        ADD HL, DE      19        ┘  add row value to it
        LD B, Ø5        Ø6 Ø5
        .....
        as before                     compute 32 * row
        .....
EOL:    DJNZ SHIFT      1Ø —
        ADD HL, DE      19            add this into HL
        ADD HL, BC      Ø9            add column value into HL
```

Now we simply execute the "draw a line" routine as before:

```
        LD A, 88        3E 88       (or whatever)
        LD B, ØA        Ø6 ØA
LOOP:   LD (HL), A      77
        INC HL          23
        DJNZ LOOP       1Ø FC
```

The hex codes are given below, tidied up.

There's no test in the routine to check that the line being drawn doesn't go over the right-hand edge of the display, and of course, such a check should be included. Otherwise a pile of end-of-line returns could get clobbered. The easiest way of doing this would be to test whether the character we're about to overwrite is a newline. If so, don't.

This routine produces a horizontal line because of the INC HL instruction in the loop. Change HL by some value other than 1, and we get different shapes. INC HL twice, and every other print position will display the character, for instance. Add 33 (decimal) into HL in every loop and we get a vertical line. Add 34 (decimal) into HL in each loop and we get a diagonal line.

You could have a library of such routines and simply call one whenever you want that kind of line.

LISTING

Here's the complete code. This time we won't bother with addresses in the listing: they're not important (thanks, once again, to *relative* jumps).

```
        LD C, ØØ        ØE ØØ
        LD E, ØØ        1E ØØ
        LD HL, (4ØØC)   2A ØC 4Ø
        INC HL          23
        LD D, ØØ        16 ØØ
        ADD HL, DE      19
        LD B, Ø5        Ø6 Ø5
SHIFT:  SLA D           CB 22
        SLA E           CB 23
        JRNC EOL        3Ø Ø1
        INC D           14
EOL:    DJNZ SHIFT      1Ø F7
        ADD HL, DE      19
        ADD HL, BC      Ø9
        LD B, ØØ        Ø6 ØØ
        LD A, ØØ        3E ØØ
LOOP:   LD (HL), A      77
        LD DE, ØØ ØØ    11 ØØ ØØ
        ADD HL, DE      19
        DJNZ LOOP       1Ø F9
```

The zero bytes underlined must be poked before calling the routine, as follows:

Start address + 1:	starting column (e.g. Ø5 for column 5)
Start address + 3:	starting row (e.g. Ø7 for row 7)
Start address + 25:	number of characters to be plotted (e.g. ØA)
Start address + 27:	code of graphics character (e.g. 86 for ▚)
Start address + 3Ø:	value added to HL between plots (e.g. Ø1 for a horizontal line, 21 for a vertical line, 2Ø or 22 for diagonal lines)
Start address + 31:	not normally used unless the value to be added exceeds 255, otherwise set to ØØ

Once you've loaded this up, and seen what it does, think about incorporating it into BASIC programs to generate, say, a series of squares (such as the RECTANGLE PLOT on page 41 of ***Timex/Sinclair 1000***). Use RND to find the top left-hand corner (column and row) and the length of side. Then POKE the relevant addresses in the machine code routine, and call it via USR. Do this four times for the four sides of the (open) rectangle. Don't forget to test the sizes to see if it will all fit on the screen!

Some new, powerful commands: block search and block transfer. The alternate registers. And programs to SCROLL limited sections of the screen, or scroll sideways, at high speed.

22 Some Things I haven't told you

First things first: actually, it was Humpty-Dumpty (see page 115).

I've deliberately simplified (even oversimplified) in places in this chapter, and I don't apologize for that. After all, machine code isn't the easiest thing to tackle, and listing all its features all at once is just confusing. On the other hand, if you've looked at other books on Z80 machine code, or ***1000*** machine code listings in magazines, you may well be wondering why I've frequently done things in an apparently cack-handed way.

So I'll try to redress the balance now.

BLOCK SEARCH

First, there are some very powerful instructions which will search a whole block of memory. I'll take CPDR which is short for "compare, decrement and repeat" as an example.

If I write a piece of code like this:

```
        LD BC, Ø1ØØ      Ø1 ØØ Ø1
        LD HL, 5ØØØ      21 ØØ 5Ø
        LD A, Ø5          3E Ø5
        CPDR              ED B9
NEXT:   — —
```

what happens is this:

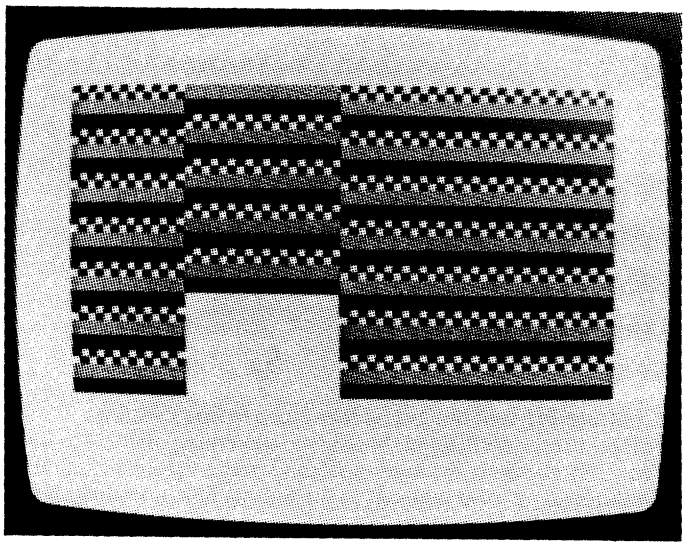
When the CPDR instruction is encountered the value in the A-register is compared with the contents of the byte HL is pointing at. If they are equal, control passes to NEXT: If not, BC and HL are both decremented by 1, and the "compare" is repeated until a match is found or until BC contains zero. In other words, those 4 instructions say: "Find the first occurrence of a byte containing Ø5 from address 5ØØØ (hex) down to address 4FØØ and leave HL pointing to it. If there isn't one, zero BC."

So the first example I gave of using jumps, which was a little "compare" loop, could have been done much more easily. But, of course, it wouldn't have illustrated jumps!

BLOCK TRANSFER AND PARTIAL SCROLLING

Secondly there are also some *block transfer* commands LDIR and LDDR which are invaluable for shifting blocks of data around in memory. For instance, to use LDIR, you:

- Load HL with the address of the first byte to be transferred.
- Load DE with the address of the first destination byte.
- Load BC with the number of bytes to be moved.



Partial scrolling.

Then LDIR transfers the first byte; increments HL and DE; decrements BC, and keeps doing this until BC hits Ø. LDDR is similar, but it decrements HL and DE (and decrements BC as before). To see LDIR in action, here's a very useful routine that lets you SCROLL a band of columns on the screen, leaving the rest fixed.

```
        COLUMN      Ø7 ØØ      set up
        WIDTH       Ø9 ØØ      data
        LD A, ØØ    3E ØØ
        LD HL, (D-FILE)  2A ØC 4Ø   get ready
        LD DE, (COLUMN)  ED 5B ØØ 7F for block
        ADD HL, DE  19             transfer
        INC HL      23
LOOP:   LD D, H     54
        LD E, L     5D
        LD BC, ØØ21 Ø1 21 ØØ   shift section
        ADD HL, BC  Ø9         of a line up
        PUSH HL     E5         to line
        LD BC, (WIDTH)  ED 4B Ø2 7F  above
        LDIR        ED BØ
        POP HL      E1         test and
        INC A       3C         repeat until
        CPA, 16     FE 16      last line
        JRNZ, LOOP  2Ø ED      reached
```

The numbers in COLUMN and WIDTH can be POKEd to change them: here I've set up a scroll starting in column 7 and of width 9, which means that columns 7, 8, 9, 1Ø, 11, 12, 13, 14, 15 will SCROLL and the rest not. There are 4 data bytes before the start address.

To see this in action, you'll need to give it something *to* scroll: a clear screen isn't too dramatic! One way to do this is to add some lines to LOADER:

```
212  FOR I = 1 TO 7
214  PRINT "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
            BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
            CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC";
216  NEXT I
```

And, for scrolling up more than one line, put the USR instruction inside a loop:

```
218  FOR I = 1 TO 13
22Ø  LET Y = USR (B + D)
222  NEXT I
```

Experiment, putting different designs on the screen before scrolling.

The underlined 16 can be changed (but only to something smaller) to scroll only down to a certain row. By changing the start, you can select a rectangle and scroll that. But, you'll need to think about what happens to the bottom line, or the scroll will just copy it out again in the same place.

You *can* write a BASIC partial scroll; but it's much too slow to be interesting.

SIDEWAYS SCROLL

Here's another useful routine in the same vein. It scrolls the whole screen sideways, as if the display were cylindrical. In each line of display the first character is pushed on to the stack, the rest shifted down, and then the first one is popped off again to the top end. This is repeated for all lines of display.

```
        LD A, Ø          3E ØØ      set loop count
        LD HL, (D-FILE)  2A ØC 4Ø   initialize
        INC HL           23         for block
        LD D, H          54         transfer of
        LD E, L          5D         first line
        INC HL           23
LOOP:   PUSH AF          F5         save loop count
        LD A, (DE)       1A
        PUSH AF          F5
        LD BC, 31 dec    Ø1 1F ØØ   rotate one
        LDIR             ED BØ      row one
        POP AF           F1         space left
        DEC HL           2B
        LD (HL), A       77
        POP AF           F1         increment
        INC A            3C         loop count
        INC HL 3 TIMES   23 23 23   adjust for
        INC DE TWICE     13 13      next row
        CP A, 22 dec     FE 16      jump unless
        JRNZ LOOP        2Ø EA      loop count is on
                                    last row
```

To see this at its best, embed it in a BASIC loop so that the screen scrolls repeatedly; and give it a nice display to scroll. For instance

```
212 FOR I = Ø TO 21
214 PRINT AT I, I; "SCROLL"
216 NEXT I
22Ø LET Y = USR (B + D)
222 GOTO 22Ø
```

Note that in machine code you can print to the 22nd and 23rd rows of screen, normally reserved for error messages; so you could change the next-to-last line of machine code to

CP A, 24 dec FE 18

Note that if you use a 24-row screen as part of a BASIC program, it is possible for the bottom two lines to interfere with the operating system and cause a crash, in some circumstances.

For yet another example of block transfer, see DOWNLOAD, Appendix 7.

MNEMONICS

The next thing I should mention is that some people use slightly different mnemonic opcodes from those I've described. For instance, where I would write LD A, (nn) some people write LD (nn). This is because the A-register is the only register which can be loaded directly, so it's not strictly necessary to specify it. I find that actually quoting it every time is a useful aid to memory, though.

ALTERNATE REGISTERS

I said that there is an alternate set of registers, and then promptly ignored them. You can always get away without using them, and they aren't very useful anyway. You can't do arithmetic in them. Their main use is to save temporarily the contents of the main set while you're executing some routine which alters the main register contents in ways you don't want. This is done by exchanging the contents of the main and alternate sets before, and again after, the offending routine:

```
EX AF, AF'      Ø8      swap AF with AF'
EXX             D9      swap BC, DE, HL with BC', DE', HL'
CALL . . .      CD — —  call offending routine
EX AF, AF'      Ø8  ┐
                    ├   restore registers
EXX             D9  ┘
```

Of course, you could do the same thing by PUSHing the register contents you want saved on to the stack before the CALL, and POPping them off afterwards.

In compute-and-display (SLOW) mode the ***1000*** operating system makes use of the A' and F' registers, so only use these if you're running in FAST. Also, the IX register is used—so for indexing use IY. In the same vein, my LOADER program assumes that it's always necessary to load the I and IY registers with 1E and 4ØØØ, as the Sinclair *Manual* says; but this is usually unnecessary, and you may be able to omit these commands. But the RET *is* necessary.

FLAGS

I steered clear of the technicalities of the F-register; but here's a brief summary: for more details consult one of the machine code books in the bibliography.

There is room in the F-register for eight flags; but it only uses six of its available bits. The flags are:

C	Carry flag
Z	Zero flag
S	Sign flag
P/V	Parity/Overflow flag
H	Half-Carry flag
N	Subtract flag

These are arranged in the register like this:

S	Z	X	H	X	P/V	N	C

where X means "not used".

The Carry flag is affected mainly by add, subtract, rotate and shift commands, and we've already seen how it works.

The Zero flag is affected by a tremendous number of commands. Roughly, if anything (except LD, INC, DEC) changes the contents of A, then the Zero flag is set (to 1) if A is zero, and reset (to Ø) otherwise. BIT sets the flag if the specified bit is zero. CP sets or resets the flag according to the result of a comparison.

The Sign flag stores the sign bit of the result of whichever operation has just been carried out: 1 for negative, Ø for positive.

The P/V flag works differently on arithmetic or logical commands. In arithmetic, it is set if there is an *overflow* in 2's complement arithmetic (e.g. if the sum of two positive numbers runs off the end of the accumulator, giving an apparently negative result). For a logical operation it is set to Ø if the byte in A has an *even* number of bits equal to 1; and to 1 if the number of bits equal to 1 is *odd*. The odd/even character of the number of bits equal to 1 is called the *parity* of the byte.

The H and N flags are used only for binary coded decimal calculations, and can be ignored.

ROM ROUTINES

I have also been guilty of reinventing the wheel now and then. The point is that the BASIC interpreter in ROM has to call on routines such as those we've developed. So why not simply *call* them, rather than write our own? In general, the answer is that it would have been much more sensible to do so, since it saves a lot of effort and, almost as important, computer memory. But—so far as *this* book is concerned, my aim has been to tell you about Z80 machine code, and avoid, as far as possible, the special features of the ***1000***. If all the examples had consisted of a series of calls to addresses in ROM you wouldn't have learnt much! Of course, to use the ROM routines, you need to know where they are. Ian Logan's book *Sinclair ZX81 ROM Disassembly* is good for this. While I'm on the subject of books, Toni Baker's *Mastering Machine Code on your ZX81* is a good read. It goes into much more detail than I have had space for here, but at a faster pace: and it makes heavy use of ROM routines.

EFFICIENT USE OF MACHINE CODE

Finally, I want to tie up two loose ends I left hanging right at the beginning. I said that there may be reasons other than BASIC's need to interpret statements each time they are executed for a machine code program to run faster than BASIC. I'll explain with an example:

BASIC		Machine code
10 FOR I = 20 TO 1 STEP −1		LD B, 14
.....	LOOP:	.....
50 NEXT I		DJNZ LOOP

In each case, every time the loop is executed a variable is decremented by 1. But that process is much more complicated in BASIC than it is in machine code. The reason is that since BASIC has to deal with decimal values some of the time, it assumes that it's doing so all the time, and so it actually subtracts 1.00000000, which is no easier than subracting 1.58712684. In fact, the procedure employed is quite complex and time-consuming. The machine code, on the other hand, uses a single, purpose-built instruction. The result is in the region of 100 times faster.

The other point I deferred was that machine code can occupy more memory than its BASIC equivalent. Here's an example which illustrates why:

BASIC		Machine code	No. of bytes	
30 IF R = P AND P = Q THEN				
LET P = W		LD HL, 5000	3	
		LD A, (HL)	1	
		LD HL, 5001	3	
		SUB A, (HL)	1	
		JRNZ NEXTBIT	2	
		LD HL, 5001	3	
		LD A, (HL)	1	
		LD HL, 5002	3	
		SUB A, (HL)	1	
		JRNZ NEXTBIT	2	
		LD HL, 5001	3	
		LD A, (5003)	3	
		LD (HL), A	1	
	NEXT BIT:		27	TOTAL

The machine code assumes that R, P, Q and W are held in the bytes 5000, 5001, 5002 and 5003, respectively. In practice it wouldn't be as simple as that because each number will occupy 5 bytes and the SUB will actually be a CALL to a floating point subtraction routine. In any event, the actual code would need at least as many, and probably more than, the 27 bytes shown. The equivalent BASIC line occupies only 18 bytes, 1 for each of the symbols (IF, =, W, AND, and so on), 4 for the line number and 1 for the line delimiter. The more complex the BASIC statement, the more memory overhead there is in the machine code version.

OTHER PLACES TO STORE MACHINE CODE

The main disadvantage with storing code above RAMTOP is that you can't save it. The advantage is that you *can* load other stuff in under it. But, to save the code, there are alternatives.

A favourite trick is to store the whole thing in a REM statement, the first line in the BASIC program. The first character after the REM has address 16514. So you write your BASIC starting with

```
1 REM XXXXXX . . . X
```

with enough Xs to hold the code; and then POKE the machine code in. I used this when telling you about CHECK IN, at the start of this section. The code is accessed by RAND USR 16514 (or LET Y = USR 16514, etc.) and can be saved; also it is not destroyed by RUN.

For your delectation, Appendix 7 gives a machine code program DOWNLOAD which, combined into HELPA, will automatically put the code into a REM statement—even in another program!

Another place to store the code is in a character string, which is easily located (via the system variable VARS) provided you make this the first variable declared—e.g. start with a big enough string array:

```
1 DIM A$ (79)          [to hold 79 bytes]
```

and then put the machine code into A$ (1), A$ (2), etc. as you build up the string. The start address is 6 + PEEK 16400 + 256 * PEEK 16401. The main disadvantage is that RUN or CLEAR will wipe out your code. And the start address can sometimes wander. On the other hand, while you can safely LIST a REM statement (even if the result makes little sense) it's wisest not to EDIT it, or try to write a new line into the same place to get rid of it. In certain circumstances (mostly if it's a long piece of code) you can cause a crash this way.

DEBUGGING

There are no built-in debugging facilities in machine code; and while HELPA will help you edit code, it isn't really designed to debug. (Nor are several other programs advertised as useful for "debugging" machine code!) Your best bet at this stage is to *dry-run* the routine using old-fashioned pencil and paper (see ***Timex/Sinclair 1000***, p. 60). Of course you can insert tracing statements into machine code, but watch out for changes in addresses and jump-sizes.

One useful (though apparently cackhanded) trick is to write your routine in BASIC first, and debug *that*: use only BASIC instructions that correspond to the machine code. (That is, *emulate* the machine code in BASIC.) It will work slowly, if at all; but it's debuggable.

For the really ambitious, this suggests a project. The idea is to improve HELPA by adding a routine SINGLE-STEP which will work through the program command by command, displaying the registers on the screen. You'll need to (*a*) write a machine code routine to PUSH all registers on to the stack, then display them on the screen in hex; (*b*) add to this a routine that looks for a keyboard input; (*c*) write between each line of machine code a CALL to this routine (use the ** delimiters in HELPA to signal where); (*d*) work out how to return to BASIC if you want.

For (*b*) you will need a ROM routine at address 02BB (called by CD BB 02) which puts into the HL register some numbers that depend on the key currently being pressed: for example key Z gives FBFE. (You now have all you need, but to save time, there's another routine at 07BD which puts the character code into HL, if the result of 02BB is first loaded into BC.) There's a lot more to it than this: I suggest you only try to write this if you've followed everything else in this book.

DRY-RUNNING A LINE-RENUMBER ROUTINE

As an example of debugging using a dry run, I'll tell you about my first stab at a routine to renumber the lines of a BASIC program, and how it got debugged. *Being* the first stab, I didn't try for anything fancy: just a routine to go through the BASIC program and number the lines 10, 20, 30, . . . going up in 10s. Once I got that working, I reckoned I could fancy it up (arbitrary start and finish, arbitrary step size, etc.).

Now to do this, you need to know how the BASIC lines are stored in memory. They occupy a block of addresses starting at 16509 and ending immediately before the value stored in D-FILE. And each line takes the following form:

NS	NJ	LJ	LS	Code for line, including tokens	ENTER

Here NS and NJ are the senior and junior bytes of the line number (in 2-byte hex) and LJ, LS are the junior and senior bytes of the number of characters in the BASIC instruction on that line. Note the order of junior and senior bytes—it's *not* an error.

For example (we'll need this for the dry-run) the program

```
1 REM
5 PRINT A
```

is stored as follows:

Address	Contents	Comments
16509	0	Senior byte of first line number
16510	1	Junior byte of first line number
16511	2	Junior byte of length of line
16512	0	Senior byte of length of line
16513	234	Token for REM
16514	118	ENTER
16515	0	Senior byte of second line number
16516	5	Junior byte of second line number
16517	3	Junior byte of line length
16518	0	Senior byte of line length
16519	245	Token for PRINT
16520	38	Code for A
16521	118	ENTER
16522	118	ENTER that starts display file
16523	0	First character in display file
.....		

You should check all this by typing in the little test program above and PEEKing 16509, etc. in command mode. Also, check that D-FILE holds 16522 by asking for PRINT PEEK 16396 + 256 * PEEK 16397.

Now, our task is to search through the BASIC area looking for line numbers, and changing them. Note that (apart from 16509–10) the line numbers are the two bytes *after* an ENTER. So the idea is to look for ENTERs, increment, and change two bytes; repeat, keep going till you reach the end of the BASIC area.

This led to the following code:

```
                LD HL, (D-FILE)         2A ØC 4Ø
                LD BC, 165Ø9            Ø1 7D 4Ø
                SBC HL, BC              ED 42
                LD B, H                 44
                LD C, L                 4D
                DEC BC                  ØB
                LD HL, 165Ø9            21 7D 4Ø
                LD DE, 1Ø DEC           11 ØA ØØ
                JR: RENUMBER            18 16
LOOP:           LD A, Ø                 3E ØØ
                CPA, B                  B8
                JRNZ SKIP               2Ø Ø5
                CPA, C                  B9
                JRNZ SKIP               2Ø Ø2
                JR END                  18 1B
SKIP:           INC HL                  23
                DEC BC                  ØB
                PUSH BC                 C5
                LD A, (HL)              7E
                LD, B, ENTER            Ø6 76
                CPA, B                  B8
                POP BC                  C1
                JRNZ LOOP               2Ø EC
FOUND:          INC HL                  23
                DEC BC                  ØB
RENUMBER:       LD (HL), D              72
                INC HL                  23
                DEC BC                  ØB
                LD (HL), E              73
                PUSH HL                 E5
                LD HL, 1Ø DEC           21 ØA ØØ
                ADC HL, DE              ED 5A
                LD D, H                 54
                LD E, L                 5D
                POP HL                  E1
                JR LOOP                 18 DB
END:            [standard ending]
```

The first block of code sets up HL to the start address 16509 and loads BC with the length of the BASIC area: to guard against trouble at the end BC is decremented here. LOOP is the dogsbody; it first checks to see if BC has got down to 0 yet; and if not, searches for an ENTER. If it finds one it passes on to FOUND, which moves HL on one place ready for RENUMBER. This changes the line number, and adds 10 to DE ready for the next line. Once BC hits 0, the routine goes to END.

Well, that was the theory. In practice, it led to a decorative crash.

So what was wrong?

To find out, I did a dry-run on the little test program. This is summarized, in a rather cryptic form, below: the idea is to track through step by step and see what happens to the registers, the stack, and the data in the BASIC area. As you work through, successive lines list how the registers change; but to save space I've put a lot of stuff on one line when I can. You'll need pencil and paper: build up the table yourself, comparing it with mine.

A	F	B	C	D	E	H	L	STACK
		16509				16522		
						13		
		13						
		12		10		16509		

[At this point there is a jump to RENUMBER. Addresses 16509 and 16510 get the new values 0, 10 . . . meanwhile the registers go like this:]

A	F	B	C	D	E	H	L	STACK
		0	11			16510		16510
						10		
				20		20		
						16510		—
0	✓							
	x	0	10			16511		0 10
2		118						
	x	0	10					—
0	✓							
	x	0	9			16512		0 9
0		118						
	x	0	9					—
0	✓							
	x	0	8			16513		0 8
234		118						
	x	0	8					—
0	✓							
	x	0	7			16514		0 7
118		118						
	✓							

[Having found an ENTER, the routine continues into FOUND and RENUMBER: addresses 16515 and 16516 get the new line number 20; and the dry-run continues:]

A	F	B	C	D	E	H	L	STACK
		0		7				—
		0		6		16515		
		0		5		16516		
						10		16516
					30	30		
						16516		—
0	✓							
	x	0		4		16517		0 4
3		118						
	x	0		4				—
0	✓							
	x	0		3		16518		0 3
0		118						
	x	0		3				—
0	✓							
	x	0		2		16519		0 2
245		118						
	x	0		2				—
0	✓							
	x	0		1		16520		0 1
38		118						
(*)	x	0		1				—
0	✓							
	x	0		0		16521		0 0
118		118						

And now (if you've followed this far!) we see we're in trouble. BC has got down to zero just too late; the program is about to wander into the *display file* and start operating on that; and it's missed the jump to END. The problem? BC was one too big to start with! If BC had been one smaller, we'd have hit zero at the line marked (*) and finished.

The remedy: decrement BC one more, between lines 7 and 8 of the machine code routine, so that DEC BC . . . 0B gets repeated. That's all!

Try it out. Load the revised routine above RAMTOP, as usual; hit NEW; write in a program with irregular line numbers like

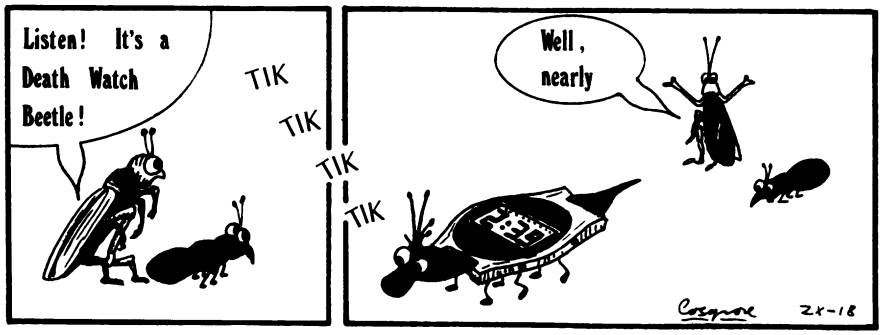
```
   73 REM
  122 PRINT P
 8001 RETURN
 8002 PLOT 3, 5
 8003 REM
 9997 STOP
```

and then ask for RAND USR 32512 (or wherever you've put it).

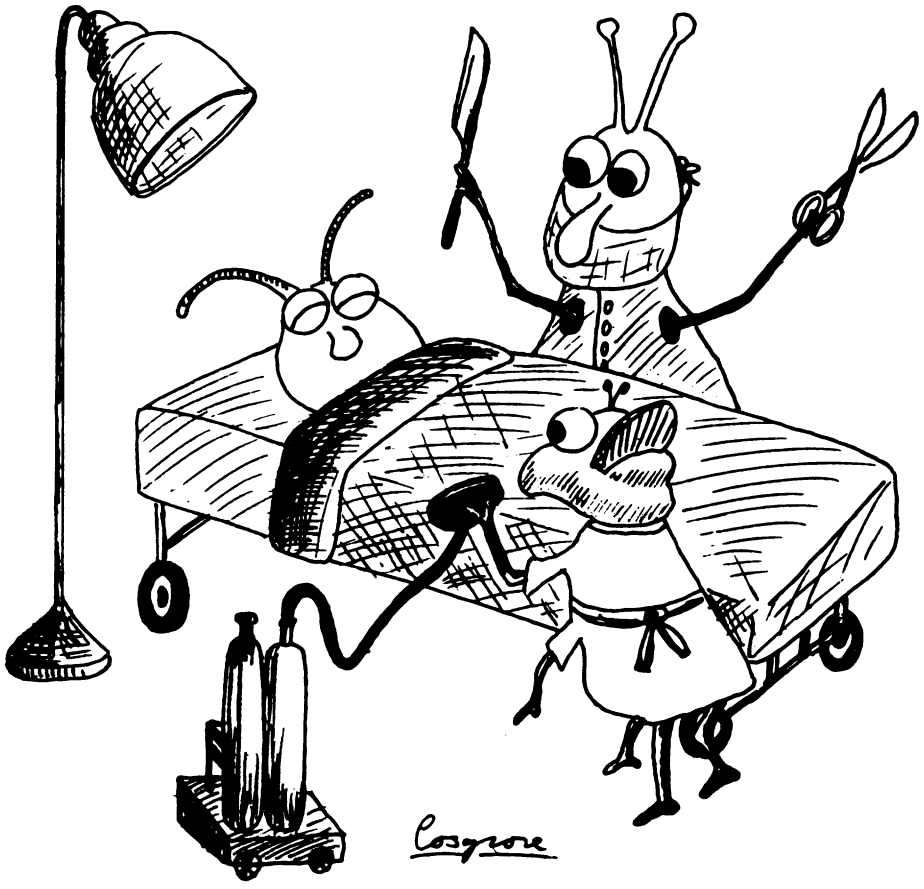
If you've got HELPA running, use it to renumber HELPA. It's amazingly fast.

As it stands, the routine isn't perfect. It doesn't do anything about GOTO or GOSUB line numbers; so after renumbering you'll have an editing job to do. There's also a more subtle bug: a long enough line can have a 118 in its "length of line" bytes; and the routine as written will go haywire here. So here are two projects: protect against the long-line bug; and work out how to deal with GOTOs and GOSUBs automatically.

A final moral: observe where the error arose. *Not* in the complicated part of the routine, where the main work was done: but in the condition for the work to end. This kind of "boundary problem" is a constant source of errors in programming. Unless you work out very carefully indeed the precise "edges" of what you're trying to do, you can easily get lost in the wilderness.



Appendices



1 Hex/Decimal Conversion

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

2's complement

ordinary

2 Memory Reservation Tables

To reserve 256-byte blocks with 16K of RAM, the addresses are as shown. Leave the contents of address 16388 at zero.

No. of 256-byte blocks	Size of area reserved	POKE 16389	Start address of area	
			Decimal	Hex
0	0	128	32768	8000
1	256	127	32512	7F00
2	512	126	32256	7E00
3	768	125	32000	7D00
4	1024	124	31744	7C00
5	1280	123	31488	7B00
6	1536	122	31232	7A00
7	1792	121	30976	7900
8	2048	120	30720	7800
9	2304	119	30464	7700
10	2560	118	30208	7600
11	2816	117	29952	7500
12	3072	116	29696	7400
13	3328	115	28440	7300
14	3584	114	29184	7200
15	3840	113	28928	7100
16	4096	112	28672	7000
17	4352	111	28416	6F00
18	4608	110	28160	6E00
19	4864	109	27904	6D00
20	5120	108	27648	6C00

3 System Variable Addresses

This table gives the hex and decimal addresses for a number of useful system variables—see the Timex *Manual* for their functions.

Variable	No. of bytes	Hex	Decimal
ERR-SP	2	4002	16386
RAMTOP	2	4004	16388
D-FILE	2	400C	16396
DF-CC	2	400E	16398
VARS	2	4010	16400
E-LINE	2	4014	16404
STKBOT	2	401A	16410
STKEND	2	401C	16412
BERG	1	401E	16414
MEM	2	401F	16415
LAST-K	2	4025	16421
SEED	2	4032	16434
FRAMES	2	4034	16436
MEMBOT	30	405D	16477
(RAMTOP usual value 1K)	2	4400	17408
(RAMTOP usual value 16K)	2	8000	32768

4 Summary of Z80 Commands

This is a list of all the opcode mnemonics, with a summary of their effects. Those commands explained more fully in the text are given a page reference. The effects on the flags are omitted: for these, consult the Zilog Reference Card or the books listed in the bibliography under "Machine Code".

ADC	Page 137	Add, including the carry flag. Store in A or HL.
ADD	Page 128	Add, ignoring the carry flag. Store in A or HL.
AND	Page 134	Logic AND on corresponding bits: store in A.
BIT	Page 144	BIT b, r sets the Zero flag according to the value of the b-th bit of the byte in register r. The bits are in the order 76543210 within each byte.
CALL	Page 118	Calls a subroutine. There are conditional calls signalled by the additional letters C (call if the Carry flag is set); M (if the Sign flag is set—"the result (of a compare) is negative"); NC (if the Carry flag is not set); NZ (if the Zero flag is not set); P (if the Sign flag is not set—"the result is positive"); PE (if the Parity flag is set: ignore this one); PO (if the Parity flag is not set: ignore this too); Z(if the Zero flag is set). For flags, see Page 134,156.
CCF		Complement Carry flag (i.e. swap 0 and 1).
CP	Page 134	Compare: sets the flags as if it were a subtraction from A, but leaves A unchanged.
CPD	Page 152	Compare and decrement. Compare through HL; then decrement HL and BC.
CPDR	Page 152	Compare, decrement, repeat: block search. Like CPD but repeating until either the result of the comparison is 0, or BC reaches 0.
CPI	Page 152	Like CPD except that HL increments; BC still decrements.
CPIR	Page 152	Like CPDR but incrementing HL.
CPL		Complement (flip bits of) the A-register.
DAA		Decimal adjust accumulator. Used in binary-coded decimal work: ignore.
DEC	Page 136	Decrement: reduce value by 1.
DI		Disable interrupts. Ignore.
DJNZ	Page 137	Decrement, jump if non-zero. Decrement B and jump relative unless the Zero flag is set. Used in loops like a BASIC FOR/NEXT.
EI		Enable interrupts. Ignore.
EX	Page 155	Exchange values. Instructions with (SP) exchange the registers HL, IX or IY with the top of the stack.
EXX	Page 155	Exchange all three register-pairs BC, DE, HL with their alternates BC', DE', HL'.
HALT		Wait for an interrupt. Unless you've got hardware attached, and know what you're doing, DO NOT USE as the program will wait forever.
IM		Interrupt mode: ignore.
IN		Input from a device. Ignore.
INC	Page 136	Increment: increase value by 1.
IND, INDR, INI, INIR		Input commands analogous to LDD, LDDR, LDI, LDIR. Ignore.
JP	Page 135	Jump. Variants with added C, M, NC, NZ, P, PE, PO, Z are conditional jumps, with conditions as for CALL.
JR	Page 135	Jump relative—followed by a 1-byte displacement. Conditional variants are C, NC, NZ, Z only.
LD	Page 126	Load. Can use all five addressing modes.
LDD		Not the same as LD D! Load what HL points to into what DE points to: decrement BC, DE, HL.

LDDR	Page 152	Load, decrement, repeat: block transfer. Do LDD until BC hits zero. Copies a block of memory whose length is stored in BC, out of what HL points to and into what DE points to.
LDI	Page 152	Like LDD except that HL and DE increment: BC still decrements.
LDIR	Page 152	Like LDDR except that HL and DE increment.
NEG		Negative: change the sign of the contents of A.
NOP		No operation. Do nothing for 1 time-cycle—i.e. waste time. Useful for temporary deletion of instructions when debugging: harmless and helpful.
OR	Page 134	Logic OR on bits. Store in A.
OTDR, OTIR, OUT, OUTD, OUTI		Various outputs. Ignore.
POP	Page 138	Pop from stack into indicated register.
PUSH	Page 138	Push from register on to stack.
RES		Reset a bit—i.e. make it zero.
RET	Page 118	Return from subroutine. Conditional returns, corresponding to the possibilities for CALL, are possible. (Conditions on a CALL need not match those on a RET!)
RETI, RETN		Return from interrupt subroutines. Ignore.
RL		Rotate left: like a shift, except that the carry flag is included as if it were bit number 8.
RLA		Rotate left accumulator. Like RL A but with a different effect on the flags.
RLC		Not the same as RL C! Rotate left, but put bit 7 into carry *and* into bit Ø.
RLCA		Like RLC A, but same flag difference as RLA.
RLD		Not what you'd expect at all: rotate left decimal. Used for binary coded decimal: ignore.
RR		Like RL but to the right.
RRA		Like RLA.
RRC		Like RLC.
RRCA		Like RLCA.
RRD		Like RLD.
RST		Like CALL, but only from addresses Ø, 8, 1Ø, 18, 2Ø, 28, 3Ø, 38 (hex). These are all in the ROM on the ZX81: see Ian Logan's books in the Bibliography. RST Ø is like temporarily disconnecting the power.
SBC	Page 137	Subtract, taking account of the carry flag. Store in A or HL.
SCF		Set carry flag (to 1).
SET		Set a bit—i.e. make it 1.
SLA	Page 137	Shift left arithmetic. All bits move up 1; bit Ø becomes Ø.
SRA	Page 137	Shift right arithmetic. Move bits down 1; copy bit 7 into 6 *and* 7.
SRL	Page 137	Shift right logical. Move bits down 1 place; make bit 7 zero.
SUB	Page 128	Subtract, ignoring carry. Store in A. (There is no SUB HL, r command: if you want one, reset the Carry flag and use SBC.
XOR	Page 134	Exclusive or on each bit. Store in A.

5 Z80 Opcodes

This is a complete list of Z80 opcodes, arranged alphabetically by mnemonic. In the listing, the symbol n stands for any single-byte number; nn for any 2-byte number; and d is 1-byte displacement written in 2's complement notation. Note that all 2-byte numbers are encoded with the junior byte *preceding* the senior.

Examples:

LD BC, nn has the opcode 01 nn, so LD BC, 732F codes as 01 2F 73.

LD A, (IY + d) " " " FD 7E d, so LD A (IY + 07) " " FD 7E 07.

The table of opcodes is based on one published by Zilog Inc. A listing numerically by opcode is included in the Timex *Manual* Appendix A: note the use of lower case letters there for mnemonics.

ADC A, (HL)	8E
ADC A, (IX + d)	DD8Ed
ADC A, (IY + d)	FD8Ed
ADC A, A	8F
ADC A, B	88
ADC A, C	89
ADC A, D	8A
ADC A, E	8B
ADC A, H	8C
ADC A, L	8D
ADC A, n	CEn
ADC HL, BC	ED4A
ADC HL, DE	ED5A
ADC HL, HL	ED6A
ADC HL, SP	ED7A
ADD A, (HL)	86
ADD A, (IX + d)	DD86d
ADD A, (IY + d)	FD86d
ADD A, A	87
ADD A, B	80
ADD A, C	81
ADD A,D	82
ADD A, E	83
ADD A, H	84
ADD A, L	85
ADD A, n	C6n
ADD HL, BC	09
ADD HL, DE	19
ADD HL, HL	29
ADD HL, SP	39
ADD IX, BC	DD09
ADD IX, DE	DD19
ADD IX, IX	DD29
ADD IX, SP	DD39
ADD IY, BC	FD09
ADD IY, DE	FD19
ADD IY, IY	FD29
ADD IY, SP	FD39
AND (HL)	A6
AND (IX + d)	DDA6d
AND (IY + d)	FDA6d
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6n
BIT 0, (HL)	CB46
BIT 0, (IX + d)	DDCBd46
BIT 0, (IY + d)	FDCBd46
BIT 0, A	CB47
BIT 0, B	CB40
BIT 0, C	CB41
BIT 0, D	CB42
BIT 0, E	CB43
BIT 0, H	CB44
BIT 0, L	CB45
BIT 1, (HL)	CB4E
BIT 1, (IX + d)	DDCBd4E
BIT 1, (IY + d)	FDCBd4E
BIT 1, A	CB4F
BIT 1, B	BC48
BIT 1, C	CB49
BIT 1, D	CB4A
BIT 1, E	CB4B
BIT 1, H	CB4C
BIT 1, L	CB4D
BIT 2, (HL)	CB56
BIT 2, (IX + d)	DDCBd56
BIT 2, (IY + d)	FDCBd56
BIT 2, A	CB57
BIT 2, B	CB50
BIT 2, C	CB51
BIT 2, D	CB52
BIT 2, E	CB53
BIT 2, H	CB54
BIT 2, L	CB55
BIT 3, (HL)	CB5E
BIT 3, (IX + d)	DDCBd5E
BIT 3, (IY + d)	FDCBd5E
BIT 3, A	CB5F
BIT 3, B	CB58
BIT 3, C	CB59
BIT 3, D	CB5A
BIT 3, E	CB5B
BIT 3, H	CB5C
BIT 3, L	CB5D
BIT 4, (HL)	CB66
BIT 4, (IX + d)	DDCBd66
BIT 4, (IY + d)	FDCBd66
BIT 4, A	CB67
BIT 4, B	CB60
BIT 4, C	CB61
BIT 4, D	CB62
BIT 4, E	CB63
BIT 4, H	CB64
BIT 4, L	CB65
BIT 5, (HL)	CB6E
BIT 5, (IX + d)	DDCBd6E
BIT 5, (IY + d)	FDCBd6E
BIT 5, A	CB6F
BIT 5, B	CB68
BIT 5, C	CB69
BIT 5, D	CB6A
BIT 5, E	CB6B
BIT 5, H	CB6C
BIT 5, L	CB6D
BIT 6, (HL)	CB76
BIT 6, (IX + d)	DDCBd76
BIT 6, (IY + d)	FDCBd76
BIT 6, A	CB77
BIT 6, B	CB70
BIT 6, C	CB71
BIT 6, D	CB72
BIT 6, E	CB73
BIT 6, H	CB74
BIT 6, L	CB75
BIT 7, (HL)	CB7E
BIT 7, (IX + d)	DDCBd7E
BIT 7, (IY + d)	FDCBd7E
BIT 7, A	CB7F
BIT 7, B	CB78
BIT 7, C	CB79
BIT 7, D	CB7A
BIT 7, E	CB7B
BIT 7, H	CB7C
BIT 7, L	CB7D
CALL C, nn	DCnn
CALL M, nn	FCnn
CALL NC, nn	D4nn
CALL nn	CDnn
CALL NZ, nn	C4nn
CALL P, nn	F4nn
CALL PE, nn	ECnn
CALL PO, nn	E4nn
CALL Z, nn	CCnn
CCF	3F
CP (HL)	BE
CP (IX + d)	DDBEd
CP (IY + d)	FDBEd
CPA	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FEn
CPD	EDA9
CPDR	EDB9
CPI	EDA1
CPIR	EDB1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX + d)	DD35d
DEC (IY + d)	FD35d
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ, d	10d
EI	FB
EX (SP), HL	E3
EX (SP), IX	DDE3
EX (SP), IY	FDE3
EX AF, AF'	08
EX DE, HL	EB
EXX	D9
HALT	76
IM 0	ED46
IM 1	ED56
IM 2	ED5E
IN A, (C)	ED78

IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn
JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD BC, (nn)	ED4Bnn
LD BC, nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED5Bnn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FD66d
LD H, A	67
LD H, B	60
LD H, C	61
LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D
LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDA0
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3
OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n
OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB82
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D

RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6
RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16
RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDCBd1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F

RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D
SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, B	CBD8
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4,(IX + d)	DDCBdE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE3
SET 4, H	CBE4

SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE
SET 5, A	CBEF
SET 5, B	CBE8
SET 5, C	CBE9
SET 5, D	CBEA
SET 5, E	CBEB
SET 5, H	CBEC
SET 5, L	CBED
SET 6, (HL)	CBF6
SET 6, (IX + d)	DDCBdF6
SET 6, (IY + d)	FDCBdF6
SET 6, A	CBF7
SET 6, B	CBF0
SET 6, C	CBF1
SET 6, D	CBF2
SET 6, E	CBF3
SET 6, H	CBF4
SET 6, L	CBF5
SET 7, (HL)	CBFE
SET 7, (IX + d)	DDCBdFE
SET 7, (IY + d)	FDCBdFE
SET 7, A	CBFF
SET 7, B	CBF8
SET 7, C	CBF9
SET 7, D	CBFA
SET 7, E	CBFB
SET 7, H	CBFC
SET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

6 HELPA

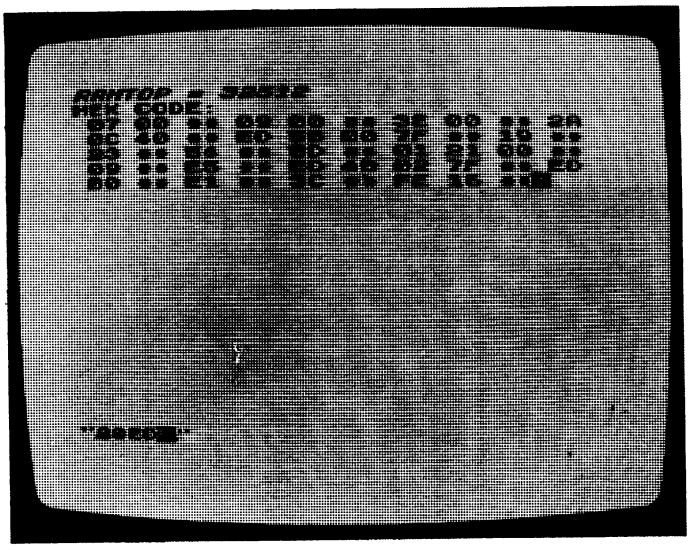
This is a versatile utility program, written in BASIC for easy modification, to help you run machine code. The acronym stands for *Hex Editor, Loader, and Partial Assembler.* A full listing is given below: here's how to use it.

1. On first use, after typing it in and debugging, SAVE using GOTO 1400 and hit NEWLINE. It will then be automatically saved under the "default" name HELPA.
2. On LOAD or RUN it will first ask whether you have reserved memory space. If so type "Y"; if not, "N". In the latter event it will now *automatically* (and without disturbing the BASIC program) reset RAMTOP to give a multiple of 256 bytes spare space, asking you for this multiple. The program will halt with a listing: ignore this and type RUN, this time answering "Y". The value now in RAMTOP will be printed. (For example, hit "N" and then "2", followed by "RUN": RAMTOP goes to 32256; see Appendix 2 for a table of values.) This resetting of RAMTOP uses the PRINT USR 1040 trick, avoiding the use of NEW.
3. It now types HEX CODE: and gives you a cursor ▷ .
4. You may now type in (and, using control commands described below, manipulate) machine code in hex. As each code group is keyed in, the program strips it of any blank spaces (which means you can use these as you want), splits it into two-character codes, and prints out these in a 10-column display. A final ** delimiter is added on the end of each group; the cursor moves to the end of the code just input.

For example, the input "□A1□E23□□4" is printed out as

A1 E2 34 ** ▷

(and so would "A1E234" or "A1□E2□34" be).



HELPA engaged in loading the Partial Scroll routine: about to input the final instruction 20 ED.

5. The next group of codes may now be input, and will be automatically appended immediately after the cursor position. The **s are for the user's convenience only and are ignored when the code is loaded above RAMTOP, so the groups do not *have* to correspond to Z80 instruction groups; however this helps for checking.
6. In addition, "control" instructions may be input. These must be the *first* symbol in their group; subsequent characters are usually ignored except in a few cases described below. These control commands are rather like ZEDTEXT mnemonics, but the codes and functions differ.

The control commands are:

G	(Go)	Run the machine code routine
L	(Load)	Load the machine code routine above RAMTOP
M	(Move)	Move cursor forward
N	(Negative)	Move cursor backward
P	(Print)	Print out current hex listing
R	(Relative)	Used for automatic calculation of relative jumps
S	(Save)	Save program
X	(eXcise)	Delete from the listing

More detailed descriptions follow in a more convenient order.

M, N The command Mn, where n is a number, moves the cursor forward n spaces. Nn moves it back n spaces. It is protected against going off the listing. If n is omitted it will be set to 1.

X A command Xn where n is a number ≥ Ø will delete the next n pairs of characters (including **s) after the cursor. The display will not be affected until P is pressed. If n is omitted it will be set to 1.

To *append* text, just set the cursor to a position immediately in front of where it should go, and enter it. Everything beyond will be shifted up to make room, but only the new text displayed until P is pressed.

P Prints the current text, and moves the cursor to the top.

S Saves the program, including the current hex listing. There is an option allowing you to name the program as you wish. To run a saved program, SAVE it *before* typing "L" and "G". Then LOAD into BASIC, type GOTO 2ØØ (*not* RUN!!), then "L", then "G".

L Loads the machine code, stripped of superfluous ** delimiters, above RAMTOP, starting at the RAMTOP value you've set; tacks on the standard ending including the final RET command.

G This runs the routine. On typing "G" the machine will ask for the number of data bytes (i.e. bytes *before* the start address) and will then run the program once this has been input. Control returns to HELPA (barring a crash!).

R A useful feature to make *relative* jumps easy: these are a nuisance by hand because of the need to count displacements and put them into 2's complement hex; but nice for programs because of portability. It works like this.

(*a*) Input the machine code routine setting all relative jump sizes to ØØ.

(*b*) To change to the correct value for a given jump, set the cursor immediately in front of the ØØ, and delete by hitting "X". Now input Rn where the number n is the (decimal) position of the *destination* byte for the jump, read off from the screen display as follows. Number the rows and columns of the hex code array starting at Ø, like this:

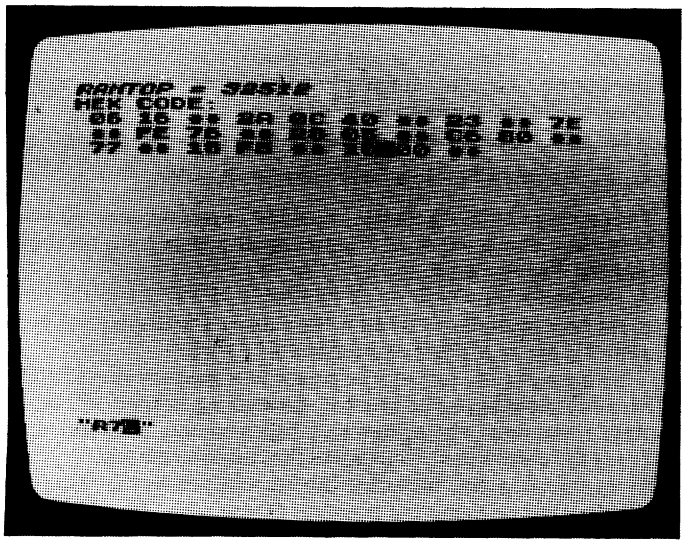
```
  Ø 1 2 3 4 5 6 7 8 9
Ø
1
2
3
.
.
.
```

and set n = xy for row x, column y. (That is, for the byte in row 17, column 5, type R175.) The fact that there are 1Ø columns makes it easy to read off the numbers. [You could modify the program so that the cursor is moved to the destination, and the jump found from that: in practice this is slower, because of all the cursor-shifting you have to do.]

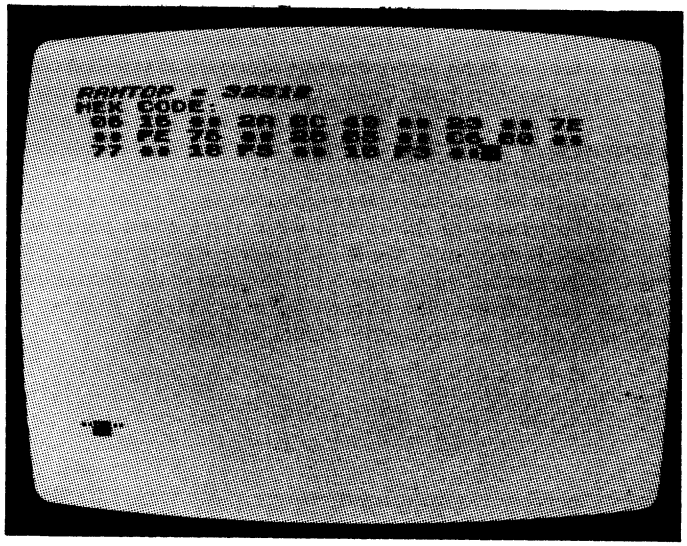
(*c*) Now press "P" for a revised display with the correct jump size included.

(*d*) Locate the cursor before the next relative jump size, and repeat.

(*e*) Note that the program automatically adjusts for any **s around and gives the 2's complement code needed: if the jump is outside permissible range, it will say so.



HELPA setting up a relative jump in the Video Inversion routine: before . . .



. . . and after.

(*f*) This may sound complicated, so here's an example, using the Video Inverter program on page 147. Set RAMTOP for one 256-byte block; input in turn the groups

Ø6 16

2A ØC 4Ø

.....

1Ø F3

but use ØØ on the JR, JRZ, and DJNZ commands. The display will be:

```
RAMTOP = 32512
HEX CODE:
Ø6 16 ** 2AØC 4Ø ** 23 ** 7E
** FE 76 ** 28 ØØ ** C6 8Ø **
77 ** 18 ØØ ** 1Ø ØØ ** ▷
```

Here the underlined ØØs are the relative jump sizes to be set. Using "N", get the cursor in front of the last ØØ like this

```
77 ** 18 ØØ ** 1Ø ▷ ØØ **
```

and then type "X" to delete the ØØ. (The screen will not change, however—yet!) The instruction here is DJNZ LOOP, and the destination LOOP starts with the code 23. This is in row Ø, column 7; so input R7. After a short pause the screen will be printed out with the new display, and the ØØ will have changed to F3, as required.

Back the cursor to the 28 ØØ; delete; input R25 (because SKIP is the group 1Ø FE starting in row 2, column 5). Now we get 28 Ø5. And so on.

Now for the program listing.

```
 5 REM [HELPA] COPYRIGHT IAN STEWART AND
   ROBIN JONES 1982.
10 PRINT "HAVE YOU RESERVED MEMORY?"
20 INPUT A$
30 IF A$ = "" THEN GOTO 50
40 IF A$ (1) = "N" THEN GOTO 9000
50 CLS
60 LET RT = PEEK 16388 + 256 * PEEK 16389
70 PRINT "RAMTOP = □"; RT
80 PRINT "HEX CODE:"
100 LET CI = Ø
110 LET H$ = ""
120 GOSUB 400
130 DIM F (20)
131 LET F (1) = 700
132 LET F (6) = 800
133 LET F (7) = 900
134 LET F (8) = 1000
135 LET F (10) = 1100
136 LET F (11) = 1200
137 LET F (12) = 1300
138 LET F (13) = 1400
139 LET F (14) = 1500
```

```
140  LET F (18) = 1600
200  INPUT I$
210  LET A = 0
220  LET A = A + 1
230  IF A > LEN I$ THEN GOTO 300
240  IF I$ (A) < > "□" THEN GOTO 220
250  LET I$ = I$ (TO A − 1) + I$ (A + 1 TO)
260  GOTO 230
300  IF CODE I$ (1) > = 44 THEN GOTO 500
310  LET I$ = I$ + "**"
320  LET H$ = H$ (TO 2 * CI) + I$ + H$ (2 * CI + 1 TO)
330  GOSUB 450
340  GOSUB 600
350  GOSUB 400
360  GOTO 200
400  REM [PRINT CURSOR]
410  PRINT AT 2 + INT (CI/10), 3 * (CI − 10 * INT (CI/10) ); "[>]";
420  RETURN
450  REM [UNPRINT CURSOR]
460  PRINT AT 2 + INT (CI/10), 3 * (CI − 10 * INT (CI/10) ); "□";
470  RETURN
500  REM [ROUTINES]
510  GOSUB F (CODE I$ (1) − 43)
520  GOTO 200
600  REM [PRINT APPENDED TEXT]
610  FOR J = 1 TO LEN I$/2
620  PRINT I$ (2 * J − 1 TO 2 * J) + "□";
630  LET CI = CI + 1
640  IF CI = 10 * INT (CI/10) THEN PRINT "□□";
650  NEXT J
660  RETURN
700  REM [GO]
710  CLS
720  PRINT "NUMBER OF DATA BYTES?"
730  INPUT D
740  CLS
750  LET Y = USR (RT + D)
760  RETURN
800  REM [LOAD]
810  LET H$ = H$ + "3E1EED47FD21004C9"
```

```
820  LET J = RT - 1
830  LET I = -1
840  LET J = J + 1
850  LET I = I + 2
860  IF I > LEN I$ THEN RETURN
870  IF H$ (I) = "*" THEN GOTO 850
880  POKE J, 16 * (CODE H$ (I) - 28) + CODE H$ (I + 1) - 28
890  GOTO 840
900  REM [MOVE CURSOR POSITIVELY]
910  GOSUB 450
920  IF LEN I$ = 1 THEN LET CM = 1
930  IF LEN I$ > 1 THEN LET CM = VAL I$ (2 TO)
940  LET CI = CI + CM
950  IF CI > LEN H$/2 THEN LET CI = LEN H$/2
960  GOSUB 400
970  RETURN
1000 REM [MOVE CURSOR NEGATIVELY]
1010 GOSUB 450
1020 IF LEN I$ = 1 THEN LET CM = 1
1030 IF LEN I$ > 1 THEN LET CM = VAL I$ (2 TO)
1040 LET CI = CI - CM
1050 IF CI < 0 THEN LET CI = 0
1060 GOSUB 400
1070 RETURN
1100 REM [PRINT]
1110 CLS
1120 PRINT "RAMTOP = □"; RT
1130 PRINT "HEX CODE:"
1140 PRINT "□";
1150 LET CI = 0
1160 FOR J = 1 TO LEN H$/2
1170 PRINT H$ (2 * J - 1 TO 2 * J) + "□";
1185 LET CI = CI + 1
1187 IF CI = 10 * INT (CI/10) THEN PRINT "□□";
1190 NEXT J
1192 GOSUB 400
1195 RETURN
1300 REM [RELATIVE JUMPS]
1310 LET JCI = VAL I$ (2 TO)
1320 LET JS = JCI - CI - 1
```

```
1195  RETURN
1300  REM [RELATIVE JUMPS]
1310  LET JCI = VAL I$ (2 TO)
1320  LET JS = JCI - CI - 1
1325  GOSUB 2000
1330  IF JS > = - 128 AND JS < = 127 THEN GOTO 1357
1335  CLS
1340  PRINT AT 10, 6; "INVALID JUMP SIZE"
1345  FOR Q = 1 TO 20
1346  NEXT Q
1350  GOSUB 1100
1355  RETURN
1357  IF JS < 0 THEN LET JS = JS + 256
1360  LET X1 = INT (JS/16)
1365  LET X0 = JS - 16 * X1
1367  LET X$ = CHR$ (X1 + 28) + CHR$ (X0 + 28)
1370  LET H$ = H$ (TO 2 * CI) + X$ + H$ (2 * CI + 1 TO)
1375  LET CI = CI + 1
1380  GOSUB 1100
1390  RETURN
1400  REM [SAVE]
1410  PRINT " [SAVE] . INPUT PROGRAM NAME.
      DEFAULT NAME HELPA."
1420  INPUT N$
1430  IF N$ = "" THEN LET N$ = "HELPA"
1440  SAVE N$
1450  CLS
1460  GOTO 10
1600  REM [DELETE]
1610  IF LEN I$ = 1 THEN LET K = 1
1620  IF LEN I$ > 1 THEN LET K = VAL I$ (2 TO)
1630  LET H$ = H$ (TO 2 * CI) + H$ (2 * CI + 2 *K + 1 TO)
1640  RETURN
2000  REM [ASTERISK ADJUST]
2010  IF JS < 0 THEN LET
      W$ = H$ (2 * JCI + 1 TO 2 * CI)
2020  IF JS > = 0 THEN LET W$ = H$ (2 * CI + 1 TO 2 * JCI)
2030  LET SC = 0
2040  FOR T = 1 TO LEN W$
2050  IF W$ (T) = "*" THEN LET SC = SC + 1
```

```
2060 NEXT T
2070 IF JS < 0 THEN LET JS = JS + SC/2
2080 IF JS > 0 THEN LET JS = JS - SC/2
2090 RETURN
9000 REM [RESERVE MEMORY]
9010 PRINT "HOW MANY 256-BYTE BLOCKS?"
9020 INPUT NB
9030 PRINT "AFTER LIST PRESS RUN"
9040 FOR T = 1 TO 20
9050 NEXT T
9060 POKE 16389, 128 - NB
9070 PRINT USR 1040
```

Remarks: In this program, the machine code is manipulated and stored in a string H$. Sections of code are input via a string I$. The cursor position is determined by CI; the position of a relative jump by JCI. The various subroutines are identified by REM statements.

7 DOWNLOAD

This is a useful addition to HELPA, which allows you to incorporate HELPA-assembled machine code into BASIC programs (which may be loaded from tape if desired) in a form which can be SAVEd. It uses a piece of machine code, also loaded above RAMTOP, which downloads your machine code into a pre-prepared REM statement in the BASIC program area. Note that *absolute* addresses will have to be changed afterwards using POKE, and are best avoided, unless you write the routine using the addresses that will be needed *after* downloading.

First, the machine code: it uses (very fast) block transfers.

CODELENGTH	— — (two bytes to be POKEd)
LD HL, (RAMTOP)	2A Ø4 4Ø
LD C, (HL)	4E
INC HL	23
LD B, (HL)	46
DEC HL	2B
LD DE, 4Ø82	11 82 4Ø
PUSH BC	C5
LD BC, 29 decimal	Ø1 1D ØØ
ADD HL, BC	Ø9
POP BC	C1
LDIR	ED BØ
LD A, 1E	3E 1E
LD I, A	ED 47
LD IY, 4ØØØ	FD 21 ØØ 4Ø
RET	C9

Build it into HELPA by adding the following BASIC program lines:

```
 141 LET F(17) = 3ØØØ
3ØØØ REM [DOWNLOAD] COPYRIGHT IAN STEWART AND
     ROBIN JONES 1982.
3Ø1Ø LET LH = Ø
3Ø2Ø FOR I = 1 TO LEN H$
3Ø3Ø IF H$(I) < > "*" THEN LET LH = LH + 1
3Ø4Ø NEXT I
3Ø5Ø LET LH = LH/2 + 9
3Ø6Ø PRINT AT 21, Ø; "LENGTH□"; LH; "□BYTES."
3Ø7Ø LET H$ = "ØØØØ2AØ44Ø4E23462B118240C5Ø11DØØØ9C1
     EDBØ3E1EED47FD21ØØ4ØC9" + H$
3Ø8Ø GOSUB 8ØØ
3Ø9Ø LET LS = INT(LH/256)
31ØØ LET LJ = LH - 256 * LS
311Ø POKE RT, LJ
312Ø POKE RT + 1, LS
313Ø RETURN
```

Use this extra facility as follows.

1. Write your machine code routine (let's name it YOURROUTINE for reference—it's the routine you want to make use of in a BASIC program, not anything to do with DOWNLOAD) using HELPA, exactly as before; but do *not* press "L" or "G".
2. Press "W" on a HELPA input, once you're happy with the code.
3. Hit NEW, *or* LOAD in your BASIC routine.
4. Make the first line of the BASIC read

    1. REM XXXXXXXX . . . XX

where the number of Xs is at least the length of YOURROUTINE (the number printed out as LENGTH . . . BYTES above). Don't forget that the "standard ending" adds 9 bytes. The routine above allows for these, but *you* may wonder where the displayed number comes from!

5. Input from the keyboard in command mode

    RAND USR (value of RAMTOP + 2).

For example, with 256 bytes reserved, this is RAND USR (32514). If you've forgotten, type

    RAND USR (PEEK 16388 + 256 * PEEK 16389 + 2).

The + 2 is crucial.

6. You will now find that line 1 of the BASIC has changed: it contains your YOURROUTINE code. You can run YOURROUTINE as a subroutine from BASIC by the command RAND USR 16514 + number of data bytes. (Incidentally those RANDS are just a convenient way to call USR, which is a *function*: LET Y = USR 16514 is just as good but uses more space and time.) The 16514 isn't magic: it's the address of the first byte after the REM in line 1. *You must make the REM statement the first in the BASIC code.* (At least, by the time you work out why this isn't quite true, you'll also know what else you can do.)
7. Now write the rest of your BASIC program around this.
8. *Warning*: You can LIST, or even LIST 1, without doing any harm—even if the REM runs over a full screen. The listing will look pretty weird, but no matter. However, do *not* attempt to erase line 1 by typing in a new line1; and don't try to EDIT line 1. You can, under certain conditions, cause a crash and have to start all over again. You can edit all other lines, though.
9. The resulting BASIC program can be SAVEd as usual, and will run perfectly . . . provided you've written it correctly!

DOWNLOAD does a rather interesting thing: it passes a routine from one program, stored on one tape, to a different program stored on another tape. It so happens this is easier in machine code than BASIC . . . but a nice project is to work out how to do the same kind of job on BASIC routines. The resetting of RAMTOP, and loading of a copy of the relevant routine above it, is the key step; then load the new one and download the routine. But, in BASIC, watch out for the operating system! The line numbers, in particular, are a problem.

Note that you can use DOWNLOAD to store *several* machine code routines in a long enough REM: just type them out end-to-end using HELPA. Press "L", the "P" after each for the standard ending. The 16514 has to change, according to which routine you want: just add 16514 to the number of bytes before the starting byte.

Bibliography

Data structures

Aho, Hopcroft and Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley.
Berztiss, *Data Structures, Theory and Practice*, Academic Press.
Brillinger and Cohen, *Introduction to Data Structures and Non-numeric Computation*, Prentice Hall.

Machine code

Baker, *Mastering Machine Code on your ZX81 or ZX80*, Interface.
Carr, *Z80 User's Manual*, Beston Publishing Co. Inc.
Logan, *Sinclair ZX81 ROM Disassembly, Parts A and B*, Melbourne House.
Logan, *Understanding your ZX81 ROM*, Melbourne House.
Nichols, Nichols and Rony, *Z80 Microprocessor Programming and Interfacing*, Howard Sams & Co.
Spracklen, *Z80 and 8080 Assembly Language Programming*, Hayden.
Zaks, *Programming the Z80*, Sybex.
Zilog Z80 CPU Programming Reference Card.
Zilog Z80 CPU Technical Manual (Zilog (UK) Ltd., Nicholson House, Maidenhead, Berks.)

General

Brady, *The Theory of Computer Science*, Chapman and Hall.
Dahl, Dijkstra, and Hoare, *Structured Programming*, Academic Press.
Sloan, *Introduction to Minicomputers and Microcomputers*, Addison-Wesley.
Tocher, *The Art of Simulation*, E.U.P.
Wegner, *Programming Languages, Information Structures, and Machine Organization*, McGraw-Hill.
Weizenbaum, *Computer Power and Human Reason*, W.H. Freeman.

Introductory

Stewart and Jones, ***Timex/Sinclair 1000: Programs, Games and Graphics***, Birkhäuser Boston.

Index

Bold type denotes especially significant references.

absolute address 182
accumulator **114**, 115
ADC 133, **137**, 169
ADD **128**, 133, 169
addition in machine code **128**–31
address **114**, 123
addressing mode 133
algorithm 2
alternate registers **125**, 155
AND 133, **134**, 169
append 53, **58**
architecture (hypothetical) 114
architecture (Z80) 124–5
arithmetic in machine code **128**–31, 141–3
array **3**–11, 15, 22, 25, 31, 41
assembler 122
assembly code 122
attic **130**, 143

base 107
binary 51, **107**
binary coded decimal 169–70
binary digit 108
binary/decimal conversion 107
bit 108
BIT **144**, 169
block search 152
block transfer **152**–4, 182
bottom line print **96**, 98
branch 31
byte 11, 51, **124**

CALL **118**, 139, 169
carry 107
carry flag **135**, 156
CHECK-IN 105
CHECKOUT 69–81
chess 44
command mode **88**, 130
compiler 68
Connect-4 44
CP 133, **134**, 169
CPDR **152**, 169
crash 57, **140**
cursor **56**, 174

data 2, 25, 28, **107**, 117
data retrieval system **29**, 32
data structure **2**, 49, 52, 69, 107
debugging 48, 55, **158**–63
decimal/hex conversion 109–10, **166**
decode 53
decrement 136
DEC 133, **136**, 169
delete 50, 53, **60**
delimiter **25**, 53, 65
dequeue **22**, 23, 70
D-FILE **145**, 148
die **72**, 81
dimension 4
direct addressing 126
direct data 87
discount 5
displacement 127, **135**, 171
display file 145–51

displaying a character 146
distribution 101
DJNZ 133, **137**, 169
documentation **65**, 80, 102
DOWNLOAD 182–3
dry-running 54, **159**–63

educational computing 82
enqueue **22**, 23, 70
EX **155**, 169
EXX **155**, 169
EZUG 48, **82**

family tree 32
field 123–4
find 61
flag **23**, 90, **134**, 156
flipping the bits **112**, 136
FRENCH COUNTDOWN 82–102

game tree 36–45
gas station simulation 81
general-purpose register 125
graphics 65, 71, 93–7, **145**

HALT 140
hand assembly 122–3
HELPA 174–81
hex 51, **108**, 127, 131
hexadecimal code 108
hex/decimal conversion 108, 110, **166**
hex loader 131
hospital bed simulation 81

immediate addressing 127
increment 136
INC 133, **136**, 169
index **114**, 123, 125, 185
index flag 123
index register **114**, 123
indexed addressing 126
indirect addressing 126
indirection 114, **120**–2, 125
indirection flag 120
initialize 16
integer BASIC 51
intelligence 45
interpret **53**, 67
interpreter 49, **67**, 68, 104

JP 133, **135**, 169
JR 133, **135**, 169
jump 118, **135**–7
junior bit **141**, 144
junior byte **127**, 159
justify 49, **63**

language **53**, 67
lap counter 22
LD **126**, 133, 169
LDDR **152**, 170
LDIR **152**, 170
leaf **31**, 37
library catalogue 27
line-drawing 148–51
line-renumbering 89, **159**–63
linked list **25**–31, 50–2
load 126–7
LOADER **128**, 131, 153
load through 126

machine code 103
machine code multiplier 141–4
machine stack 139
Manual 51, 127, 129, 145, 155, 171
mask 142
memory 3, **114**
memory reservation tables 167
memory saving **88**–9, 101, 145
memory test 61
menu **29**, 91, 110
mnemonic 56, **116**, 155
module 83
mug-trap **93**, 101

named subroutine 19
negative number 111–3
Nim 36–43
node **31**–2, 36–7
number 107–13

offset 148
opcode 115, **171**–3
operation code 115
OR 133, **134**, 170
Othello 44
outline flowchart 85
overflow **131**, 156
overflow flag 156

page 58
parity 156
partial scrolling 152–4
PC **114**, 125
petrol station simulation 81
pirating 102
pointer **6**, 13, 20–2, 25, 32–5, 41, 50–1, 69, 119
pop **16**, 18–9
POP 133, **138**, 170
printer 49, 99
procedure 84
Procrustean assignment 51
program counter 114, **116**, 125
program design 83
pseudo-operation 122
push **16**, 18–9
PUSH 133, **138**, 170

queue **20**–4, 69, 79
queue array 69

RAMTOP **130**, 146
register 114
register-to-register addressing 127
relative jump **135**, 175
REM statement storage 105, **158**, 182–3
resetting RAMTOP 130–2
RET **118**, 139, 170
rocket 86, 93
ROM routine **132**, 140, 156
root **31**, 37
running machine code 130

saving 64
SBC 133, **137**, 170
search **12**–4, 33, 42, 50
search, binary 12
search, linear 12
senior byte 125, 127, **159**
shift 50, **137**
shift right arithmetic 138
shift right logical 138
sideways scrolling 154
sign flag **135**, 156
simulation **69**, 80
SLA 133, **137**, 170
slice 52
SP **114**, 125
special-purpose register 114–5, **125**
SRA 133, **137**, 170
SRL 133, **137**, 170
stack **15**–9, 118–9, 138
stack pointer 15, **16**, **114**, 125
standard ending 129
stock control 12, 30
store 115
store through 121
storing machine code **130**, 158
story-board 86
strategy **44**, 72, 77
string 3, **52**, 53
structured programming 29, **47**
SUB **128**, 133, 170
sub-menu 29
subroutine 42, 48, 53, 55, 70–3, **83**
subroutine, closed 83
subroutine, machine code 118–9
subroutine, open 83
subscript 4
suite 83
supermarket simulation 69
symbolic address 121
system variable 67, 74, **168**

table 6
teacher transparency 89
text-editor 49
top-down programming 83
Tower of Hanoi 17
tree 31–45

user-friendly 64
USR 130

value of a move **37**, 40–4
VARS 158
vector 6
video-inversion 146–7
vide-printer 23

weather map 7
word **111**, 116
word-processor 27, **49**
word size 111

XOR 133, **134**, 170

zero flag **135**, 156
ZEDTEXT **49**–68, 89
Z80 microprocessor **104**, 113, 124, 133
Z80A microprocessor 104
Z80 opcodes **133**–9, 169–70, **171**–3

2's complement notation **112**–3, 127, **166**
16-bit quirk of Z80 139

Other titles of interest

Timex Sinclair 1000
Programs, Games, and Graphics
by Ian Stewart and Robin Jones

"Far and away the best book for ZX81 [TS 1000] users new to computing."
— *Popular Computing*

Control Your TRS80
Better BASIC and Machine Code
by Ian Stewart and Robin Jones

Available in the Fall, 1983—

Introducing the TS 2000
by Ian Stewart and Robin Jones

Introduction to the Commodore 64
A Complete Guide to Understanding Your New Computer
by Nevin Scrimshaw

Easy Programming with TI BASIC
by Cortland Shurtliff

Discover the VIC-21/VIC-20
An All-Purpose Guide to Your Personal Computer
by Cortland Shurtliff

In their first book, **Timex Sinclair 1000: Programs, Games, and Graphics,** Ian Stewart and Robin Jones wrote one of the most charming and humorous introductory computer books available. Now, in **Machine Code and Better BASIC**, this crack computer team embarks on yet more adventurous forays into the computer universe.

Although written for the TS 1000 and TS 2000, the principles of good programming are machine-independent, and almost everything in this book applies to any machine built around a Z80 or Z80A CPU: the Sharp MZ80B or MZ80K, the Tandy TRS80, and the ZX81. **Machine Code and Better Basic** introduces structured BASIC programming. It also allows you to go outside BASIC altogether, with Machine Code, the language of the Z80 microchip. And one of the bonuses of Machine Code is that it teaches you more about the way the computer actually works.

Some programs include:

- searching a stock list
- enqueuing and dequeuing data
- a complete word processor
- even a test for French vocabulary

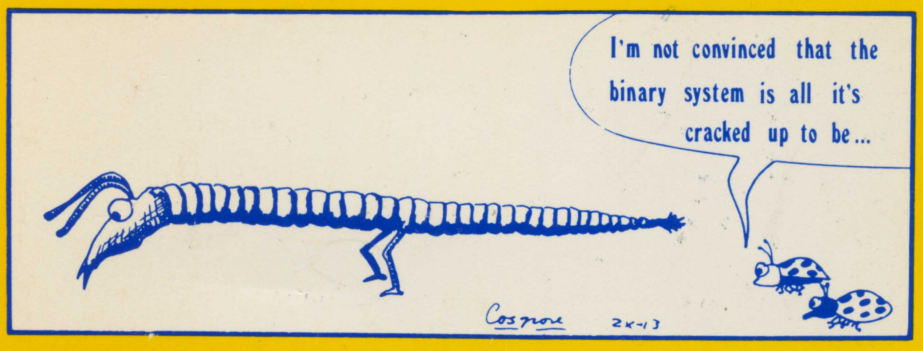
. . . and more.

There are Machine Code routines to:

- give the display a checkerboard pattern
- turn it into inverse video
- add and multiply numbers
- move data around in RAM

. . . and even more.

With this book in hand, you'll be ready to *show the computer who's boss!*



ISBN 3-7643-3115-1

$11.95

Machine Code and better BASIC

Ian Stewart and Robin Jones